

A PROGRAMMATIC INTERPRETATION OF COMBINATORY LOGICS

A THESIS

Presented to

The Faculty of the Division of Graduate
Studies and Research

by

Jorge Baralt-Torrijos

In Partial Fulfillment

of the Requirements for the Degree


Doctor of Philosophy

in the School of Information and Computer Science

Georgia Institute of Technology

June, 1973

A PROGRAMMATIC INTERPRETATION OF COMBINATORY LOGICS

Approved: 

Chairman: Lucio Chiaraviglio

Member: Gordon Pask

Member: William I. Grosky

Date approved by Chairman: June 1, 1973

This dissertation is dedicated with love
to Mayela, Libsen and Igor for their
patience and understanding during the years
of confinement which was inflicted on them
while I was engaged in this research.

ACKNOWLEDGMENTS

To begin, I would like to thank Professor Vladimir Slamecka for providing me the opportunity of becoming part of the research community in which I developed my dissertation. Also, my sincere thanks are extended to the members of my reading committee, Professors Gordon Pask and William J. Grosky for their invaluable comments on my work, and to Professors Thomas G. Windeknecht, John M. Gwynn, and Frederick A. Rossini for serving on the examining committee. Mr. John Gehl deserves my thanks for his help in the proof-reading of the final draft of this thesis. In addition, I am grateful to numerous faculty members of Universidad Simón Bolívar, Universidad Central de Venezuela, and Georgia Institute of Technology for their continuous encouragement.

Finally, and most especially, I want to thank my thesis advisor, Professor Lucio Chiaraviglio, for his expert guidance and valued friendship. I will always be grateful to Professor Chiaraviglio for having introduced me to the terrors and delights of logical reasoning.

My doctoral work was initially supported by Cia. Shell de Venezuela, Ltd., and later by Universidad Simón Bolívar and by UNESCO under the project VEN-31. This support is gratefully acknowledged.

TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS	iii
SUMMARY	v
Chapter	
I. INTRODUCTION	1
Motivation	
Objective	
Relevant Research	
II. COMBINATORY CALCULI	6
Morphology	
Theory Proper	
III. COMBINATORY LOGICS	34
Referential Interpretation	
Programmatic Interpretation	
IV. CONCLUDING REMARKS	48
Conclusions	
Future Research	
BIBLIOGRAPHY	52

SUMMARY

The objective of this dissertation is to develop a formal semantic theory for a programmatic interpretation of a wide range of combinatory calculi. The morphology of a family of combinatory calculi is presented. The notion of a tree of addresses for the representation of constructions of combinatory terms is introduced, and within that context such classical notions of combinatory logic as combination and substitution are reviewed. The theory proper of the calculi is described, and the notions of contraction, immediate reduction, weak reduction, convertibility, and extensional equivalence are introduced. The most important theorem of the first part of the dissertation is one concerning the inverse of substitution.

The semantic framework of the calculi is specified, and the calculi then become logics. The semantic completeness (in the referential sense) of these logics is proven. The programmatic interpretation is defined and the following notions are introduced: goal; process; success of a process at attaining goals; and co-success of processes in a goal language. Programmatic models are formally defined for combinatory logics and the notion of programmatic equivalence among combinatory terms is defined. Finally, the dissertation establishes those requirements which the goal language should satisfy in order for the semantic notion of programmatic equivalence to correspond with the syntactic notion of convertibility.

CHAPTER I

INTRODUCTION

The development of computers and of programming languages has been largely independent of the evolution of the mathematical theories of computability and automata. Lately, however, there is an increasing awareness by some computer scientists that abstract theories are needed for the further development of their field. At the same time, mathematicians and logicians have become interested in obtaining representations of ordinary computing machines and programming languages in some of the available formal theories.

Motivation

It may be considered that a theory is a system of assertions about objects which requires a language for its formulation, where a language is understood to mean a system of signs and of rules for their use. When the object of study of a theory S is a formal language L , then S is called the semiotics of L ; L itself is called the object language; and the language L' , in which S is formulated, is called the metalanguage.

In every situation in which a language is used, three features can be identified: the expressions of the language; the objects designated by the expressions of the language; and the users of the language. Thus, within the semiotics of such a language, three regions may be distinguished according to which of the mentioned

features receive attention: the syntax, which studies only the expressions; the semantics, which concerns itself only with the relationships between the expressions and the objects they designate; and the pragmatics, which concerns itself with the users of the language and which may include historical, sociological and psychological considerations.

The semiotic specification of a language L is provided by a syntactic framework, a semantic framework based on the syntactic framework, and a pragmatic framework based in turn on the semantic framework and determined by a set of rules of use. This dissertation is restricted to the semantic level, and no pragmatic considerations are countenanced.

The syntactic framework consist of two parts: the morphology, which determines the expressions in the language by means of a vocabulary and a grammar; and the theory proper, which provides a proof mechanism. A formal language for which only a morphology is specified is called a syntactic system. If, in addition, a theory proper is specified, then the language is called a calculus. If, finally, a semantic framework is determined, then the syntactic system becomes a semantic system, and the calculus becomes a logic.

Semantic systems, and syntactic systems with non-formal rules of interpretation, are used both for the description of objects and properties of objects (in the case when they are first order languages) and for the description of properties of properties of objects, etc. (in the case when they are higher order languages). Logics, and calculi with non-formal rules of interpretation, not only describe objects and their properties but also provide the tools which can use

assertions about objects to derive the proof of other assertions.

Ordinary programming languages are syntactic systems with non-formal rules of interpretation. However, they are weak in the sense that all of their expressions are only terms used for the description of objects; as a result, statements about the objects cannot be formulated within the same language. The terms of programming languages are called programs, and the objects denoted by them are called procedures. Two of the properties about procedures that it is desirable to be able to formulate are: correctness, (which applies if a procedure solves a problem) and equivalence (which applies if two procedures solve the same class of problems).

A programming language could be extended by including in its morphology the predicators required for the formulation of the properties mentioned above, but it will be useless to do this if a proof mechanism is not available. However, since the statements to be proven relate to procedures rather than to programs, the appropriate semantic framework may also be provided.

An alternative approach is to extend an existing logic so that it will be adequate for the representation of procedures and their properties. This extension is what is called a programmatic extension of the logic.

Objective

The objective of this dissertation is to develop a formal semantic theory for a programmatic interpretation of a wide range of combinatory calculi.

A family of combinatory calculi is specified by a schematic presentation of their morphology and theory proper. A referential semantic framework is defined for that family of calculi, and it is shown that the calculi are semantically complete in the referential sense. The intuitive notion of problem is identified with the formal notion of goal; the intuitive notion of procedure is identified with the formal notion of process-generating agent; and programs are identified with terms. The formal concepts of success and failure at attaining goals are introduced, and a formal semantic notion of programmatic equivalence is constructed on the basis of the mentioned concepts. Finally, this dissertation establishes those requirements which the goal language should satisfy in order for the semantic notion of programmatic equivalence to correspond with the syntactic notion of convertibility.

Relevant Research

Developments in the theory of computability resulted in the emergence of new areas of study related to the three fundamental entities in computing: algorithms, programs and computers. These areas are known as, respectively, the theory of recursive functions, the theory of formal languages, and the theory of finite automata. Since these theories were born from theoretical developments, they each take a theoretical approach, and the applicability of their results is not always evident. McCarthy (1962) makes a clear statement of the need for a theory of computation oriented to the solution of practical and real problems rather than theoretical and hypothetical

problems. Again, within such a theory of computation, three areas of specialization can be distinguished; they may be called the theory of algorithms, the theory of programming languages, and the theory of computing machines. For a description of the general developments in the area of programming languages, the survey papers of De Bakker (1970), Wegner (1972a), and Elspas et al. (1972) are recommended.

The works that are more closely related to the investigation here reported are: Landin (1965), Orgass (1967), and Petznick (1970) with respect to the relationship between combinatory logic and programming languages; Wegner (1972b), Manna (1969), and Luckham et al. (1970) with respect to program correctness and program equivalence; Van Fraassen (1971), Bell and Slomson (1969), and Robinson (1965) with respect to the referential interpretation; DeMillo (1972) with respect to the programmatic interpretation of programming languages; and, of course, the works of Curry and Feys (1958) and Curry, Hindley and Seldin (1972).

CHAPTER II

COMBINATORY CALCULI

In this chapter a family of systems of combinatory logic is formally specified. They are called calculi rather than logics because only the formalization of their syntax is defined and nothing is mentioned about their semantics; they are called calculi rather than syntactic systems because not only their morphology is presented but also their theory proper. In order to be able to describe a family of calculi, and not merely some particular calculus, the syntax is presented in a schematic form.

Morphology

The morphology of the combinatory calculi comprises a vocabulary that specifies the primitive symbols and a grammar that specifies the set of rules of formation for terms and formulae.

Vocabulary

Atoms. The set of atoms A of a combinatory calculus CC is the union of a non-empty finite set of individual constants C and a denumerable set of individual variables V .

Functors. CC can be minimally defined with only two functors: the corner operator ' \neg ' and the equal-double-dot predicator ' $.=.$ '. However, in order to simplify the proof of some theorems on semantics, the following predicators are also included: ' $>$ ', ' \rightarrow ', ' \Rightarrow ', ' \Leftarrow '. The set of predicators is denoted by P .

Grammar

Terms. The set of terms T of CC is defined as the smallest set satisfying:

- (i) Every atom is a term;
- (ii) If t_0 and t_1 are terms then ' $t_0 \neg_{t_1}$ ' is a term.

Formulae. The set of formulae F of CC is defined as the smallest set that satisfies

- (i) If t_0 and t_1 are terms, then ' $t_0 > t_1$ ' is a formula;
- (ii) If t_0 and t_1 are terms, then ' $t_0 \rightarrow t_1$ ' is a formula;
- (iii) If t_0 and t_1 are terms, then ' $t_0 \Rightarrow t_1$ ' is a formula;
- (iv) If t_0 and t_1 are terms, then ' $t_0 \Leftrightarrow t_1$ ' is a formula;
- (v) If t_0 and t_1 are terms, then ' $t_0 \equiv t_1$ ' is a formula.

Constructions

Let us assume that no atom is of the form

$$t_0 \neg_{t_1} \quad (1)$$

Then, from the definition of term, it may be deduced that every term that is not an atom (a non-atomic term) is of the form (1). In that case t_0 is called the functional component of t and t_1 is called the argument component of t .

Let id be the identity function from the set of terms into itself; and let 0 and 1 be partial functions from terms into terms such that for all t , if t is a non-atomic term, then $0(t)$ is the functional component of t and $1(t)$ is its argument component; otherwise,

0 and 1 are undefined. Then the set of addresses Σ of CC can be defined as the smallest set that contains the identity function and every finite composition of the functions 0 and 1. Thus, for any address σ in Σ there is a finite, possibly empty, string of 0's and 1's that names it according to the following rules:

- (i) The null string names the identity function;
- (ii) If the string X names the function σ then $0X$ names the function $0.\sigma$ and $1X$ names the function $1.\sigma$.

In order to illustrate this, the following example may be considered: let 011 be the name of the address σ , then $\sigma(t) = 0(1(1(t)))$, if defined, denotes the functional component of the argument component of the argument component of t .

If the functional component of a term t is not an atom, then t may be represented by

$$00(t) \begin{array}{c} \text{---} \\ | \quad | \\ 10(t) \quad 1(t) \end{array} \quad (2)$$

If the argument component of a term t is not an atom then t may be represented by

$$0(t) \begin{array}{c} \neg \\ | \\ 01(t) \neg \\ | \\ 11(t) \end{array} \quad (3)$$

Combining (2) and (3), any term can be represented by a particular type of tree structure. Thus

$$000(t) \begin{array}{c} \text{---} \\ | \quad | \quad | \quad | \\ 0100(t) \neg \quad 0010(t) \neg \quad 1(t) \\ | \quad | \quad | \quad | \\ 1100(t) \quad 1010(t) \quad 110(t) \end{array} \quad (4)$$

may be an acceptable representation for some term t .

If there is an address σ such that for some terms t_0 and t_1 , $t_0 = \sigma(t_1)$, then t_0 is said to be a component of t_1 , and if σ is different from the identity function then t_0 is called a proper component of t_1 .

If σ_0 and σ_1 are two different addresses, but are such that $t_0 = \sigma_0(t_1) = \sigma_1(t_1)$, then it can be said that there are at least two occurrences of t_0 in t_1 , one in address σ_0 and the other in address σ_1 .

For every occurrence of a term t_0 in another term t_1 at some address σ , a finite sequence of terms t'_0, t'_1, \dots, t'_n can be constructed such that $t_0 = t'_0$ and $t_1 = t'_n$ and for all $0 \leq i < n$ either $t'_i = 0(t'_{i+1})$ or $t'_i = 1(t'_{i+1})$. Such a sequence is called the composition of t_1 from t_0 , and it is completely determined by σ .

If the set of all atomic components of a term t is called the support of the term, then the construction of t may be defined as the set of all compositions of t from every occurrence of the members of the support of t . A natural way to represent constructions is by means of binary trees, such that every node in the tree corresponds to an occurrence of some component (the left subtree of the node corresponding to the functional component of the term in the node and the right subtree corresponding to the argument component of the term in the node). This can be illustrated by the following example.

If t is the term

$$\begin{array}{c}
 a_0 \\
 \hline
 a_1 \quad a_1 \quad a_2 \\
 \hline
 a_2 \quad a_2 \quad a_3
 \end{array} \quad (5)$$

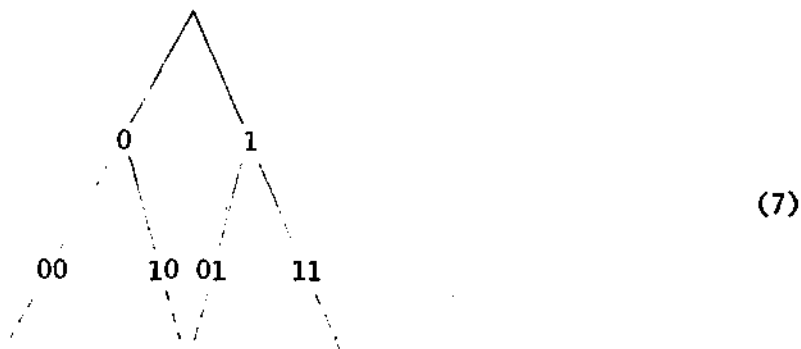
and a_0, a_1, a_2, a_3 are atoms, then, the construction of t may be represented by the binary tree.

(6)

If the relation \leq_0 is defined on the set Σ of addresses, such that for any $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$

- (i) $0 \leq \sigma_1 \leq \sigma_1$
(ii) $1 \leq \sigma_1 \leq \sigma_1$
(iii) If $\sigma_1 \leq \sigma_2$ and $\sigma_2 \leq \sigma_3$ then $\sigma_1 \leq \sigma_3$

Since $\sigma_1 \neq \sigma_2$ implies that for any σ_3, σ_4 in Σ , $\sigma_3 \cdot \sigma_1 \neq \sigma_4 \cdot \sigma_2$, then $<_0$ is a strict partial order relation that may be represented by the binary tree



This tree is called the tree of addresses of CC.

A cut K of the tree of addresses may be defined as a subset of Σ such that for all $\sigma_0 \in \Sigma$ either $\sigma_0 \in K$ or there is some $\sigma_1 \in K$ such that either $\sigma_0 <_0 \sigma_1$ or $\sigma_1 <_0 \sigma_0$; and for all $\sigma_1, \sigma_2 \in K$ neither $\sigma_1 <_0 \sigma_2$ nor $\sigma_2 <_0 \sigma_1$. Only finite cuts are considered in this investigation.

If K is a cut, then the top of K is the set of all addresses σ_0 such that there is some $\sigma_1 \in K$ and $\sigma_1 <_0 \sigma_0$; and the bottom of K is the set of all addresses σ_0 such that there is some $\sigma_1 \in K$ and $\sigma_0 <_0 \sigma_1$. If K_0 and K_1 are two different cuts, then K_0 is said to be smaller than K_1 , $K_0 <_1 K_1$, if and only if the top of K_0 is a proper subset of the top of K_1 . If $KN = \{K_0, K_1, \dots, K_n\}$ is a finite set of cuts and UK is the union of their tops, and if SK is the union of the sets $(K_i - UK)$ for all K_i in KN , then it follows immediately that SK is a cut greater than any K_i different from SK , SK is called the supremum cut of KN .

When the construction of some term t is represented upon the tree of addresses such that in the node corresponding to the address σ the term $\sigma(t)$ is placed, then only a finite number of addresses

will correspond to a defined term, since every term has only a finite number of components. The rest of the addresses remain undefined.

If all the addresses of some cut K are defined when the construction of some term t is represented in the tree of addresses, then K is called a level of resolution of t . If K_0 and K_1 are two different levels of resolution of t and if $K_0 <_1 K_1$ then K_1 is said to be a higher level of resolution than K_0 . Since the set of all the levels of resolution of t is finite, then it has a supremum; moreover, since every member of that supremum is a member of some level of resolution, the supremum is also a level of resolution. This highest level of resolution is called the boundary of the term. Such boundary uniquely determines the structure of the term; that is, the set of all defined addresses. Two terms are said to be structurally identical if and only if they have the same boundary.

The members of any cut K are well ordered by a relation $<_k$, defined naturally as follows:

If σ_1 and σ_2 are any two addresses in the cut K , then $\sigma_1 <_k \sigma_2$ if and only if the name of σ_1 is XOZ and the name of σ_2 is YlZ for some strings of 0's and 1's X, Y, Z . If σ_1 is not of the form XOZ then it means that either σ_1 is the identity function or the name of σ_1 is a string of 1's only; in the first case σ_1 is the only element of K and in the second case σ_1 is the last element of K (that is, every element in K different from σ_1 has the form XOZ). A similar analysis can be made if σ_2 is not of the form YlZ . Therefore, $<_k$ well-orders K .

If K is the boundary of some term t , then the sequence of atoms corresponding to the members of K and ordered by the relation $<_k$ is called the frame of t . Two terms are frame identical if and only if they have the same frame. Two terms are identical if and only if they are structural and frame identical.

For example, the term in (5) can be analyzed as follows:

- (a) The boundary of t is the set
 $\{000, 0100, 1100, 0010, 1010, 110, 1\}$ that uniquely
 defines the structure in (4);
- (b) The frame of t is the sequence $\langle a_0, a_1, a_2, a_1, a_2, a_3, a_2 \rangle$;
 and
- (c) The support of t is the set $\{a_0, a_1, a_2, a_3\}$.

Combinations

If B is a set of atoms and t is a term such that its support is a subset of B , then t is called a combination of B and B is called a basis for t . If the support of t is the set B , then t is called a proper combination of B .

The construction of any combination t from a basis B involves the following elementary operations: first, the cancellation of all those atoms in B that are not in the support of t ; second, the reproduction of those atoms that have more than one occurrence in t ; third, the rearrangement of the atoms to form the frame of the term; and finally, the application of the corner operator to the elements on the frame to form the term. If the combination is proper, then no cancellation is required. Cancellation, duplication, permutation and

composition are called the elementary combinatory operations.

Substitution

So far, no distinction has been made between variables and constants, and every assertion has used the more general notion of atom. When these two types of atoms are differentiated, the following definitions can be stated:

If t is a term, then the set of constants in the support of t is called the fixed part of t and the set of variables in the support of t is called the non-fixed part of t . The constant space of t is that set of all addresses in the boundary of t that contains a constant, and the variable space of t is the set of all addresses in the boundary of t that contain a variable. It can be observed that the constant and variable spaces of t form a partition of the boundary of t .

A term that does not have non-fixed part is called a closed term, otherwise it is called an open term. The set of all closed terms of CC is denoted by T_c . An open term that does not have a fixed part is called a free term. If t is either a closed term or an explicit term and v is a variable that is not a component of t , then $t \rightarrow_v$ is an explicit term. The terms that are not closed, free, or explicit are called implicit terms. The explicit terms with only one constant component are called primitive terms.

The non-fixed part of a formula is the union of the non-fixed parts of the terms in the formula. It is called a closed formula if and only if it has no non-fixed part. The set of all closed formulae is denoted by F_c .

A formula ' $t_0 > t_1$ ' is said to be a primitive formula if and only if t_0 is a primitive term and t_1 is a combination of the non-fixed part of t_0 .

The existence of variables in a calculus is not justified if the notion of substitution does not exist. In this thesis such a notion is not included in the syntax but is formulated as a metatheoretic concept. Thus, suppose that t_0 and t_1 are terms and v is a variable in V , then $[t_1/v] t_0$ denotes the term that results from the substitution of t_1 for v in every occurrence of v in t_0 . This operation is defined by the following rules.

- (S1) $[t/v] v = t$ for any $t \in T$ and $v \in V$
- (S2) $[t/v] a = a$ for any $t \in T$, $a \in A$, and $a \neq v$
- (S3) $[t/v] (t_0 \neg_{t_1}) = [t/v] t_0 \neg_{[t/v] t_1}$ for any $t_0, t_1 \in T$
- (S4) $[t/v] (t_0 > t_1) = [t/v] t_0 > [t/v] t_1$
- (S5) $[t/v] (t_0 \rightarrow t_1) = [t/v] t_0 \rightarrow [t/v] t_1$
- (S6) $[t/v] (t_0 \Rightarrow t_1) = [t/v] t_0 \Rightarrow [t/v] t_1$
- (S7) $[t/v] (t_0 \Leftrightarrow t_1) = [t/v] t_0 \Leftrightarrow [t/v] t_1$
- (S8) $[t/v] (t_0 \dot{=} t_1) = [t/v] t_0 \dot{=} [t/v] t_1$

As an immediate consequence of (S1), (S2), and (S3) the following propositions can be formulated.¹

- (S9) $[t_0/v_0] t$ is uniquely defined
- (S10) $[v_0/v_0] t = t$ for any $v_0 \in V$
- (S11) If v_0 does not occur in t then $[t_0/v_0] t = t$
- (S12) If v_0 and v_1 are distinct variables and either v_1 does

¹Curry and Feys (1958), pp. 205-209.

not occur in t_0 or v_0 does not occur in t then

$$[t_0/v_0] [t_1/v_1] t = [[t_0/v_0] t_1/v_1] [t_0/v_0] t$$

As a corollary of (S12) we have:

(S13) If v_0 and v_1 are distinct variables and v_1 does not occur in t_0 and v_0 does not occur in t_1 then

$$[t_0/v_0] [t_1/v_1] t = [t_1/v_1] [t_0/v_0] t$$

A particular case of this is when t_0, t_1 are closed terms.

Sometimes $[t_0 t_1 \dots t_n/v_0 v_1 \dots v_n] t$ or $[t_i/v_i] t$ are used as abbreviations for $[t_0/v_0] [t_1/v_1] \dots [t_n/v_n] t$

Theory Proper

Following Curry's notation, the theory proper is the part of the syntax that describes the axioms and the rules of transformation.

Axioms

Two types of axioms are considered in the definition of any combinatory calculus: the axioms of reduction that are characteristic of every calculus, and the axioms of reflexivity described by a schema that is the same for all combinatory calculi with the same morphology.

Axioms of Reduction. Every combinatory calculus is characterized by a set of primitive formulae R such that

(RED) If $r \in R$ then $\vdash r$

Axioms of Reflexivity. For every term t in T

(REF) (i) $\vdash t \Rightarrow t$

(ii) $\vdash t \Leftrightarrow t$

(iii) $\vdash t =. t$

Rules of Transformation

Again, two types of rules of transformation are considered:
the theorem-preserving rules and the rules of inference.

Theorem-Preserving Rules. For every formula $f \in F$, every term $t \in T$, and every variable $v \in V$ then

(SBT) If $\vdash f$ then $\vdash [t/v] f$.

This rule is called the rule of substitution and is the only theorem-preserving rule considered.

Rules of Inference. In order to simplify the presentation of this rule, the metatheoretic predicator variable 'p' is used in the formulation of the specific rule and followed by an enumeration of the predicators that satisfy the rule. Thus, for any terms $t_0, t_1, t_2 \in T$:

(RMN) If $\vdash (t_0 \text{ p } t_1)$ then $\vdash (t_2 \underset{t_0}{\neg} \text{ p } t_2 \underset{t_1}{\neg})$

This is called the rule of right monotony and it is satisfied by the predicators denoted by \rightarrow , \Rightarrow , \Leftrightarrow , and $\dot{=}$.

(LMN) If $\vdash (t_0 \text{ p } t_1)$ then $\vdash (t_0 \underset{t_2}{\neg} \text{ p } t_1 \underset{t_2}{\neg})$

This is called the rule of left monotony and it is satisfied by the predicators denoted by \rightarrow , \Rightarrow , \Leftrightarrow , and $\dot{=}$.

(TRN) If $\vdash (t_0 \text{ p } t_1)$ and $\vdash (t_1 \text{ p } t_2)$ then $\vdash (t_0 \text{ p } t_2)$

This is called the rule of transitivity and it is satisfied by the predicators denoted by \Rightarrow , \Leftrightarrow , and $\dot{=}$.

(SYM) If $\vdash (t_0 \text{ p } t_1)$ then $\vdash (t_1 \text{ p } t_0)$

This is called the rule of symmetry and it is satisfied by the predicates denoted by \Leftrightarrow and $\dot{=}$.

If there exists a term t_3 that is not a component of neither t_0 nor t_1 then

(EXT) If $\vdash (t_0 \dot{=}_{t_3} t_1 \dot{=}_{t_3})$ then $\vdash (t_0 \dot{=} t_1)$.

This is called the rule of extensionality.

Reduction

An occurrence t_0 of a proper component of a term $t \in T$ is in a functional position if and only if it is in some address whose name is of the form OX for any string X of 0's and 1's, and it is in argument position otherwise. If the address of t_0 in t is a member of the boundary of t , then t_0 is called a leading element of t , and in particular, if t_0 is the first member of the frame of t , then it is called the head of t . It is evident that every leading element of t is the head of at least one component of t .

The name of the address of every leading element t_0 of a term t is of the form $0^n X$, where 0^n is a non-null string of n 0's, and X is either the null string or any string of 0's and 1's starting at the left with a 1. In this case, n is called the degree of t_0 in t and X is the address of a component of t called the component of t led by t_0 . The terms in addresses whose names are of the form $10^k X$, for $0 \leq k < n^1$, are called the arguments of t_0 in t and the set of all of them is called the environment of t_0 in t . The arguments of the head of a term are called the main arguments of the term.

¹ 0^0 is the null string

The head of the primitive term of any axiom of reduction is called a combinator because it may be interpreted as an operator that acts upon a set of variables, its environment, to produce a combination of them. Since this interpretation is traditional in the studies of combinatory logic and is fundamental for the objectives of this thesis, calculi in which there are two different axioms of reduction with the same combinator are not considered. Therefore, there is a one-to-one correspondence between the set of combinators, called the combinatory base of CC, and the set R of axioms of reduction. Then, it can be said without ambiguity that the presentation pattern of a combinator is the primitive term of its corresponding axiom of reduction, and that the combination pattern of a combinator is the free term of its corresponding axiom of reduction.

If the combinatory base of CC is equal to its set of constants C , then the calculus is said to be pure combinatory, otherwise it is called an illative combinatory calculus. Although many of the works related to this dissertation have used illative concepts, this investigation is restricted to the case of pure combinatory calculi. And wherever the word "combinatory" is used henceforth in this thesis, it means pure combinatory.

Contraction. Every axiom of reduction, or every formula that results from the application of the rule of substitution to some axiom of reduction, is called a reduction rule. If $t_0 > t_1$ is a reduction rule then t_0 is called the redex of the rule and t_1 is called the contractum of the rule; and the replacement of a redex

by its contractum is called a contraction.

The predicator ' $>$ ' characterized by the axioms of reduction and the rule of substitution may be viewed as a partial function that maps terms into terms. It is partial because there are terms that are not redexes, and it is a function as a consequence of the assumption made that no redex may have two different contracta. This function may also be called contraction. Thus, a contraction may be redefined as the replacement of some term by its image under the contraction function, if the function is defined for that term.

The following lemma is entered for the sake of completeness but is not used in the sequel. Readers may skip this lemma.

Lemma 1. If f is a formula for the form ' $t_0 > t_1$ ' and v_1, v_2, \dots, v_n are the variables in the non-fixed part of f , then there are closed terms $t'_1, t'_2, t'_3, \dots, t'_n$, such that if f is not a theorem then

$$[t'_1, t'_2, \dots, t'_n / v_1, v_2, \dots, v_n] f$$

is not a theorem.

Proof: If $t_0 > t_1$ is not a theorem, then either t_0 is not a redex, or t_1 is not the contractum of the redex t_0 .

Case 1. If t_0 is not a redex, then either the head of t_0 is not a combinator or the degree of the head of t_0 is not equal to the order of the combinator that leads t_0 .

Subcase 1.1. If the head of t_0 is not a combinator then either t_0 is a variable or t_0 is a non-atomic term. If t_0 is a variable then for any constants c_1, c_2, \dots, c_n

$$[c_1 c_2, \dots, c_n/v_1, v_2, \dots, v_n] f$$

is not a theorem, since

$$[c_1, c_2, \dots, c_n/v_1, v_2, \dots, v_n] t_0$$

is a constant and no constant can be a redex. If t_0 is a non-atomic term but its head is a variable v_i then for any t'_1, t'_2, \dots, t'_n such that t'_j is a constant for all $j \neq i$ and t'_i is a closed redex

$$[t'_1, t'_2, \dots, t'_n/v_1, v_2, \dots, v_n] f$$

is not a theorem, since the degree of the head of the term

$$[t'_1, t'_2, \dots, t'_n/v_1, v_2, \dots, v_n] t_0$$

is greater than the order of the combinator that leads the term.

Subcase 1.2. If the degree of the head of t_0 is not equal to the order of the combinator that leads it, then for any constants c_1, c_2, \dots, c_n

$$[c_1, c_2, \dots, c_n/v_1, v_2, \dots, v_n] f$$

is not a theorem, since substitution by atoms does not modify the boundary of the terms and therefore does not modify the degree of their leading elements.

Case 2. If t_1 is not the contractum of the redex t_0 and q_0 denotes the presentation pattern of the combinator leading t_0 , and q_1 denotes the combination pattern of the same combinator, then either the boundary of q_1 is not a level of resolution of t_1 , or it is, but there exist addresses σ_0 and σ_1 such that σ_0 is in the variable space of q_0 and σ_1 is in the boundary of q_1 and such that $\sigma_0(q_0) = \sigma_1(q_1)$ but $\sigma_0(t_0) \neq \sigma_1(t_1)$.

Subcase 2.1. If the boundary of q_1 is not a level of resolution of t_1 then for any constants c_1, c_2, \dots, c_n

$$[c_1, c_2, \dots, c_n/v_1, v_2, \dots, v_n] f$$

is not a theorem, since substitution by atoms does not modify the boundary of the terms and therefore the boundary of q_1 is not a level of resolution of

$$[c_1, c_2, \dots, c_n/v_1, v_2, \dots, v_n] t_1$$

either.

Subcase 2.2. If the boundary of q_1 is a level of resolution of t_1 and there exist addresses σ_0, σ_1 such that σ_0 is in the variable space of q_0 and σ_1 is in the boundary of q_1 and $\sigma_0(q_0) = \sigma_1(q_1)$ but $\sigma_0(t_0) \neq \sigma_1(t_1)$, then either $\sigma_0(t_0)$ and $\sigma_1(t_1)$ are not structural identical or they are structural identical but not frame identical. If $\sigma_0(t_0)$ and $\sigma_1(t_1)$ are not structural identical then for any constants c_1, c_2, \dots, c_n ,

$$[c_1, c_2, \dots, c_n/v_1, v_2, \dots, v_n] f$$

is not a theorem, since substitution by atoms does not modify boundaries, and therefore it does not modify structures either. If $\sigma_0(t_0)$ and $\sigma_1(t_1)$ are structural identical but not frame identical then there must be an address σ_2 in their boundary such that $\sigma_2(\sigma_0(t_0)) \neq \sigma_2(\sigma_1(t_1))$. If $\sigma_2(\sigma_0(t_0))$ and $\sigma_2(\sigma_1(t_1))$ are two different constants then for any closed terms t'_1, t'_2, \dots, t'_n ,

$$[t'_1, t'_2, \dots, t'_n/v_1, v_2, \dots, v_n] f$$

is not a theorem, since constants are not modified by substitution.

If $\sigma_2(\sigma_0(t_0))$ is a constant c and $\sigma_2(\sigma_1(t_1))$ is a variable v_i then for any closed terms t'_1, t'_2, \dots, t'_n such that $t'_i \neq c$,

$$[t'_1, t'_2, \dots, t'_n/v_1, v_2, \dots, v_n] f$$

is not a theorem, since the frames of the components of the resulting terms in the addresses σ_0 and σ_1 respectively are not equal. The case in which $\sigma_2(\sigma_1(t_1)) = c$ and $\sigma_2(\sigma_0(t_0))$ is a variable is the same case as the preceding one. Finally, if $\sigma_2(\sigma_0(t_0))$ is the variable v_1 and $\sigma_2(\sigma_1(t_1))$ is the variable v_j then for any closed terms t'_1, t'_2, \dots, t'_n , if $t'_i \neq t'_j$ then

$$[t'_1 t'_2 \dots t'_n / v_1, v_2, \dots, v_n] f$$

is not a theorem by the same reason as in the previous three cases.

Immediate Reduction. It may be the case that a term is not a redex, but some component of it is indeed a redex. If it is desirable to be able to replace every component of a term that is a redex by its contractum then the rules of left and right monotony should be added to contraction. The predicate ' \rightarrow ' is called immediate reduction.

It is said that t_0 immediately reduces to t_1 , $t_0 \rightarrow t_1$, if and only if t_1 results from the contraction of exactly one of the redexes in t_0 . If t_0 does not have redexes, then it is called a normal form, and it cannot immediately reduce to any term.

Null redexes. A null redex is a redex that contracts to itself. The redexes in the axioms of reduction cannot be null, since no primitive term can be a free term. Therefore, null redexes can only be obtained by substitution. Thus, if

$$U \frac{}{t_1 t_2 \dots t_n} > U \frac{}{t_1 t_2 \dots t_n} \quad (8)$$

then the axiom of reduction corresponding to the combinator U must be of the form

$$U \begin{array}{c} \hline v_1 \quad v_2 \quad \dots \quad v_j \quad \dots \quad v_n \end{array} > v_j \begin{array}{c} \hline v_1 \quad v_2 \quad \dots \quad v_j \quad \dots \quad v_n \end{array} \quad (9)$$

Hence, the combinator U must have a duplicative effect in the sense that the variable v_j occurs twice in the combination pattern of U.

The simplest null redex is the one generated by the axiom of reduction

$$U \begin{array}{c} \hline v_1 \end{array} > v_1 \begin{array}{c} \hline v_1 \end{array} \quad (10)$$

that by substitution generates the rule of reduction

$$U \begin{array}{c} \hline U \end{array} > U \begin{array}{c} \hline U \end{array} \quad (11)$$

Lemma 2. If f is a formula of the form ' $t_0 \rightarrow t_1$ ' and v_1, v_2, \dots, v_n are the variables in the non-fixed part of f , then there are closed terms t'_1, t'_2, \dots, t'_n such that if f is not a theorem then

$$[t'_1, t'_2, \dots, t'_n / v_1, v_2, \dots, v_n]f$$

is not a theorem.

Proof: Let σ, σ' be addresses in the set of addresses Σ of the combinatory calculus, and let $<_0$ the partial ordering that generates the tree of addresses. Let $\text{Redex}(\sigma(t))$ be true if and only if $\sigma(t)$ is a redex, and $\text{Var}(\sigma(t))$ be true if and only if $\sigma(t)$ is a variable. Let $[/]$ be the representation of the substitution

$$[t'_1, t'_2, \dots, t'_n/v_1, v_2, \dots, v_n].$$

By definition of immediate reduction

$$\vdash (t_0 \rightarrow t_1) \text{ iff } (\exists \sigma)(\text{Redex}(\sigma(t_0)) \ \& \ (\sigma')(\sigma' \neq \sigma \ \& \ \sim(\sigma' <_0 \sigma) \supset \sigma'(t_0) = \sigma'(t_1)) \\ \& \ \vdash (\sigma(t_0) > \sigma(t_1)))$$

that is

$$\sim \vdash (t_0 \rightarrow t_1) \text{ iff } (\sigma)(\text{Redex}(\sigma(t_0)) \supset ((\exists \sigma')(\sigma' \neq \sigma \ \& \ \sim(\sigma' <_0 \sigma) \ \& \\ \sigma'(t_0) \neq \sigma'(t_1)) \vee \sim \vdash (\sigma(t_0) > \sigma(t_1))))$$

and also

$$\sim \vdash ([/]t_0 \rightarrow [/]t_1) \text{ iff } (\sigma)(\text{Redex}(\sigma([/]t_0)) \supset ((\exists \sigma')(\sigma' \neq \sigma \ \& \\ \sim(\sigma' <_0 \sigma) \ \& \ \sigma'([/]t_0) \neq \sigma'([/]t_1)) \vee \sim \vdash (\sigma([/]t_0) > \sigma([/]t_1))))$$

To prove that

$$\sim \vdash (t_0 \rightarrow t_1) \supset \sim \vdash ([/]t_0 \rightarrow [/]t_1)$$

is the same as proving that

$$(\sigma)(\text{Redex}(\sigma(t_0)) \supset ((\exists \sigma')(\sigma' \neq \sigma \ \& \ \sim(\sigma' <_0 \sigma) \ \& \ \sigma'(t_0) \neq \sigma'(t_1)) \vee \\ \sim \vdash (\sigma(t_0) > \sigma(t_1))))$$

implies

$$(\sigma)(\text{Redex}(\sigma([/]t_0)) \supset ((\exists \sigma')(\sigma' \neq \sigma \ \& \ \sim(\sigma' <_0 \sigma) \ \& \ \sigma'([/]t_0) \neq \\ \sigma'([/]t_1))) \vee \sim \vdash (\sigma([/]t_0) > \sigma([/]t_1))))$$

and this is an immediate consequence of proving that

$$a) \quad \text{Redex}(\sigma([/]t_0)) \text{ iff } \text{Redex}(\sigma(t_0)) \vee \text{Var}(\sigma(t_0))$$

$$b) \quad \text{Var}(\sigma(t_0)) \supset \sim \vdash (\sigma([/]t_0) > \sigma([/]t_1))$$

- c) $(\exists \sigma')(\sigma' \neq \sigma \ \& \ \sim (\sigma' <_0 \sigma) \ \& \ \sigma'(t_0) \neq \sigma'(t_1)) \supset$
 $(\exists \sigma')(\sigma' \neq \sigma \ \& \ \sim (\sigma' <_0 \sigma) \ \& \ \sigma'([/] t_0) = \sigma'([/] t_1))$
- d) $\text{Redex}(\sigma(t_0)) \ \& \ \sim \vdash (\sigma(t_0) > \sigma(t_1)) \supset \sim \vdash (\sigma([/] t_0) >$
 $\sigma([/] t_1))$

In order to satisfy a) above, the terms t'_1, t'_2, \dots, t'_n in the substitution should be redexes.

In order to satisfy b), the redexes selected should be such that if one is a substitute for a variable in some address then its contractum should not be identical to the component in that address after substitution has been made. To satisfy c) and d) it is sufficient that

$$(\sigma)(\sigma \in \Sigma \ \& \ \sigma(t_0) \neq \sigma(t_1) \supset \sigma([/] t_0) \neq \sigma([/] t_1))$$

The only assertion that remains without proof is that for any combinatory calculus a set of closed terms t'_1, t'_2, \dots, t'_n that satisfy a) through d) exists.

In effect, any combinatory calculus has at least one constant, and therefore at least one axiom of reduction. By the rules of formation, a denumerable infinite set of closed terms can be generated from the set of constants; therefore, substitution by closed terms on the axioms of reduction generates a denumerable infinite set of closed terms that are redexes. From this initial set of eligible redexes are excluded those which are components of either t_0 or t_1 ; since there are only a finite number of them, the resulting set of eligible redexes is still denumerable infinite. The null redexes are also excluded to satisfy condition b) above in the case that

$\sigma(t_0) = \sigma(t_1)$; the resulting set is again denumerable infinite, since for every null redex that is generated from some axiom of reduction there is at least one non-null redex generated from the same axiom. All those redexes whose contractum is a component of t_1 are also excluded; since there are only a finite number of them, the resulting set of eligibles remains denumerable infinite. From this set the terms t'_1, t'_2, \dots, t'_n can be selected in a form that conditions a) through d) above are satisfied.

Combinatorial Completeness

It is often necessary to analyze the relationship between terms that have been obtained by successive contractions from other terms. To do that, transitivity is added to immediate reduction; if reflexivity is also included, then a partial order relation among terms (called reduction and characterized by the predicator \Rightarrow) is obtained. Sometimes this relation is called weak reduction to distinguish it from strong reduction which in addition includes the rule of extensionality. In this thesis the distinction is unnecessary since strong reduction is not considered.

A combinatory calculus is said to be combinatorial complete if and only if for every term t and for every variable v there is a term, denoted by $[v]t$ and called the functional abstraction of t with respect to v , such that v is not a component of $[v]t$ and

$$[v]t \underset{\lambda}{\rightarrow} t \Rightarrow t \quad (12)$$

Indeed, it is always the case that if the calculus is combinatorial complete, not one but many terms satisfy the conditions for functional abstraction of some term t and variable v . Therefore, in order to assure the uniqueness of the notation $[v] t$, an algorithm should be provided to produce the functional abstraction given the term and the variable. Such algorithms depend on the combinatory base of the calculus but are not unique for that calculus; examples of such algorithms can be found in Curry and Feys.¹ The existence of an algorithm is a sufficient condition for the combinatorial completeness of a calculus.

The notation for functional abstraction can be generalized for the case of more than one argument; thus $[v_1, v_2, \dots, v_n] t$ stands for $[v_1] [v_2] \dots [v_n] t$, such that

$$[v_1, v_2, \dots, v_n] t \frac{}{v_1 \ v_2 \ \dots \ v_n} \Rightarrow t \quad (13)$$

It may be observed that for any free term t in a combinatorial complete calculus, if v_1, v_2, \dots, v_n are the variables in the support of t , then the formula (13) resembles an axiom of reduction. Therefore, if the appropriate considerations are made, then in order to mirror any calculus, it is sufficient to have a combinatory base with the minimal number of combinators required for combinatorial completeness.

Lemma 3. If f is a formula of the form ' $t_0 \Rightarrow t_1$ ' and v_1, v_2, \dots, v_n are the variables in the non-fixed part of f ,

¹Curry and Feys (1958), pp. 190-194.

then there are closed terms t'_1, t'_2, \dots, t'_n such that if f is not a theorem then

$$[t'_1, t'_2, \dots, t'_n / v_1 v_2, \dots, v_n] f$$

is not a theorem.

Proof: The procedure used in the proof of lemma 2 can be extended to the case of more than two terms; that is, to the case of finite chains of immediate reductions such as

$$q_0 \rightarrow q_1 \rightarrow \dots \rightarrow q_n$$

In such case

$$\sim \vdash (q_0 \rightarrow q_1) \supset \sim \vdash ([/]q_0 \rightarrow [/]q_1)$$

and

$$\sim \vdash (q_1 \rightarrow q_2) \supset \sim \vdash ([/]q_1 \rightarrow [/]q_2)$$

and so on.

Then

$$\sim \vdash (q_0 \rightarrow q_1) \vee \sim \vdash (q_1 \rightarrow q_2) \vee \dots \vee \sim \vdash (q_{n-1} \rightarrow q_n)$$

implies

$$\sim \vdash ([/]q_0 \rightarrow [/]q_1) \vee \sim \vdash ([/]q_1 \rightarrow [/]q_2) \vee \dots \vee \sim \vdash ([/]q_{n-1} \rightarrow [/]q_n)$$

and since

$$t_0 \neq t_1 \supset [/]t_0 \neq [/]t_1$$

Then

$$\sim \vdash (t_0 \Rightarrow t_1) \supset ((q_0)(q_1) \dots (q_n)(q_0 = t_0 \ \& \ q_n = t_n \supset (\sim \vdash (q_0 \rightarrow q_1)$$

$$\vee \dots \vee \sim \vdash (q_{n-1} \rightarrow q_n) \ \& \ t_0 \neq t_1)$$

and

$$\sim \vdash (t_0 \Rightarrow t_1) \supset ((q_0)(q_1) \dots (q_n)(q_0 = [/]t_0 \ \& \ q_n = [/]t_n \supset$$

$$(\sim \vdash ([/]q_0 \rightarrow [/]q_1) \vee \dots \vee \sim \vdash ([/]q_{n-1} \rightarrow [/]q_n))) \ \& \ [/]t_0 \neq [/]t_1)$$

and

$$\sim \vdash (t_0 \Rightarrow t_1) \supset \sim \vdash ([/]t_0 \Rightarrow [/]t_1)$$

Combinatory Equivalence

In many interpretations of combinatory calculi, a partial ordering is not a strong enough relation between terms because equivalence must be countenanced. In this thesis two types of equivalence relations among terms are considered: convertibility and extensional equivalence.

Convertibility. This relation is obtained by including the rule of symmetry with the rules that define reduction. This relation is characterized by the predicator ' \Leftrightarrow ', and its most important property is the so-called Church-Rosser property, which says:

If $t_0 \Leftrightarrow t_1$ then there is a t_2 such that

$$t_0 \Rightarrow t_2 \text{ and } t_1 \Rightarrow t_2 \tag{14}$$

Since the right-to-left implication in (14) is immediate by transitivity and symmetry, then convertibility can be defined as follows: t_0 and t_1 are convertible if and only if there is some term t_2 to which both t_0 and t_1 reduce.

Lemma 4. If f is a formula of the form ' $t_0 \Leftrightarrow t_1$ ' and v_1, v_2, \dots, v_n are the variables in the non-fixed part of f , then there are closed terms t'_1, t'_2, \dots, t'_n such that if f is not a theorem then

$$[t'_1, t'_2, \dots, t'_n / v_1, v_2, \dots, v_n] f$$

is not a theorem.

Proof: In the same form that lemma 3

$$\begin{aligned} \sim \vdash (t_0 \Leftrightarrow t_1) &\supset (t_2) (\sim \vdash (t_0 \Rightarrow t_2) \vee \sim \vdash (t_1 \Rightarrow t_2)) \\ &\supset (t_2) (\sim \vdash ([/]t_0 \Rightarrow [/]t_2) \vee \sim \vdash ([/]t_1 \Rightarrow [/]t_2)) \\ &\supset \sim \vdash ([/]t_0 \Leftrightarrow [/]t_1) \end{aligned}$$

Extensional Equivalence. This relation among terms is obtained by adding the rule of extensionality to convertibility. It is characterized by the predicate ' $=$ ' and it is the most powerful relation of the combinatory calculi studied in this investigation.

Lemma 5. If f is a formula of the form ' $t_0 = t_1$ ' and v_1, v_2, \dots, v_n are the variables in the non-fixed part of f , then there are closed terms t'_1, t'_2, \dots, t'_n such that if f is not a theorem, then

$$[t'_1, t'_2, \dots, t'_n / v_1, v_2, \dots, v_n] f$$

is not a theorem, provided the combinatory calculus is combinatorial

complete.

Proof: This is done by proving that if for all closed terms t'_1, t'_2, \dots, t'_n ,

$$[t'_1, t'_2, \dots, t'_n/v_1, v_2, \dots, v_n]f$$

is a theorem then f is a theorem; and this is proven by induction.

Initial Case: If for all closed terms t'_1 , $[t'_1/v_1] f$ is a theorem then f is a theorem.

If f is of the form $t_0 = t_1$ and t'_1 is any closed term then

- a) $[t'_1/v_1] t_0 = [t'_1/v_1] t_1$ by hypothesis
- b) $[v_1] t_0 \neg_{t'_1} = [v_1] t_1 \neg_{t'_1}$ by combinatorial completeness and (SBT)
- c) $[v_1] t_0 = [v_1] t_1$ by extensionality.
- d) $[v_1] t_0 \neg_{v_1} = [v_1] t_1 \neg_{v_1}$ by left monotony
- e) $t_0 = t_1$ by definition of functional abstraction.

Induction Case: Let us suppose that if for all closed terms t'_1, t'_2, \dots, t'_n ,

$$[t'_1, t'_2, \dots, t'_n/v_1, v_2, \dots, v_n] f$$

is a theorem then f is a theorem. Then prove that if for all closed terms $t'_1, t'_2, \dots, t'_n, t'_{n+1}$,

$$[t'_1, t'_2, \dots, t'_n, t'_{n+1}/v_1, v_2, \dots, v_{n+1}] f$$

is a theorem then f is a theorem.

$$\vdash [t'_1, t'_2, \dots, t'_n/v_1, v_2, \dots, v_n][t'_{n+1}/v_{n+1}]f$$

by hypothesis

$$\vdash [t'_{n+1}/v_{n+1}][t'_1, t'_2, \dots, t'_n/v_1, v_2, \dots, v_n]f$$

by property of substitution

$$\vdash [t'_1, t'_2, \dots, t'_n/v_1, v_2, \dots, v_n]f$$

by initial case

$$\vdash f \quad \text{by hypothesis of induction}$$

Inverse of Substitution

A predicator p is said to have an inverse of substitution in a combinatory calculus, if and only if for any terms t_0 and t_1 if v_1, v_2, \dots, v_n are the variables in their non-fixed part, and for any closed terms t'_1, t'_2, \dots, t'_n

$$[t'_1, t'_2, \dots, t'_n/v_1, v_2, \dots, v_n] (t_0 \text{ p } t_1) \supset \vdash t_0 \text{ p } t_1 \quad (15)$$

Theorem 1. All the predicators in the combinatory calculi here described have inverse of substitution.

Proof: The proof is an immediate consequence of lemmas 1 to 5.

CHAPTER III

COMBINATORY LOGICS

In this chapter a semantic framework for the combinatory calculi described in the previous chapter is specified. The systems here presented are called combinatory logics in the sense that their syntax as well as their semantics has been completely formalized. No confusion should arise with the traditional connotation of combinatory logic as the study of functional application and functional abstraction, where most of the primitive ideas have a fixed interpretation.

Referential Interpretation

An interpretation of a calculus is a correspondence between the formulae of the calculus and certain statements which are significant without reference to the calculus. Curry and Feys¹ call the latter statements *contentive* statements. A calculus may have an interpretation in another calculus or it may have a completely intuitive interpretation. A particular type of interpretation, called *referential interpretation*,² assigns to the terms in the

¹Curry and Feys, (1958) p. 21.

²Van Fraassen, (1971) p. 107.

calculus elements in some domain of discourse, and to the predicates in the calculus relations defined in the domain of discourse. In this form, the contentive statements are membership statements; that is, statements asserting that an element is or is not a member of some set.

Realizations and Models

A realization of a combinatory logic is an ordered pair $\langle D, \delta \rangle$ such that:

- a) D is a non-empty set called the domain of discourse of the realization.
- b) δ is a function called the interpretation function of the realization, and it is defined by:
 - (i) for every constant c , $\delta(c) \in D$,
 - (ii) for every predicate p , $\delta(p) \subseteq D^2$,
 - (iii) $\delta(\neg): D^2 \rightarrow D$.
- c) Γ is the set of all assignment functions γ , such that $\gamma: V \rightarrow D$.
- d) The interpretation of any term t for some assignment γ , denoted by $t[\gamma]$ is determined by the rules
 - (i) If t is a constant then $t[\gamma] = \delta(t)$,
 - (ii) If t is a variable then $t[\gamma] = \gamma(t)$
 - (iii) If t is a non-atomic term then

$$t[\gamma] = \delta(\neg) \langle 0(t)[\gamma], 1(t)[\gamma] \rangle$$

It may be observed that if t is a closed term then $t[\gamma]$ is the same for any assignment γ , in this case the notation $\delta(t)$ may be used

instead of $t[\gamma]$ to emphasize the fact that the interpretation of such a term is independent of the assignment.

- e) The satisfaction of a formula f by some assignment γ , in a realization R , is denoted by $R \models f[\gamma]$ and is determined by the rule

$$R \models (t_0 \text{ p } t_1) [\gamma] \text{ iff } \langle t_0 [\gamma], t_1 [\gamma] \rangle \in \delta(p)$$

for any predictor p .

If a formula f is satisfied in a realization R , by every assignment, $R \models f$, then f is said to be valid in R . R is said to be a model of a combinatory logic CL if and only if every theorem of CL is valid in R . The set of all models of CL is denoted by $STR(CL)$.

Semantic Completeness

A combinatory logic CL is semantic complete if and only if every formula of CL that is valid in all models of CL is a theorem of CL .

Theorem 2. Every combinatory logic specified by the syntactic and semantic frameworks described above is semantic complete.

Proof: This theorem is proven by showing that if CL is the combinatory logic in question then for every formula f of CL that is not a theorem there is a model of CL , and an assignment of that model for which f is not satisfied.

It is evident that a free model of CL is the model desired. This model is constructed as follows:

- a) The domain of discourse is the set of all atoms of CL and all the tree-like structures obtained from the infixing of the corner sign ' \neg ' between any two terms;
- b) $\delta(\neg)$ is the operation of infixing the corner sign between any two terms to form a new term;
- c) $\delta(p)$ for any predictor p is the set of all ordered pairs $\langle t_0, t_1 \rangle$ such that $t_0 p t_1$ is a theorem of CL.

Thus, by construction, if ' $t_0 p t_1$ ' is not a theorem then $\langle t_0, t_1 \rangle \notin \delta(p)$, and therefore it is not satisfied by an assignment γ such that, for any variable v , $\gamma(v) = v$.

Since every combinatory logic has a free model then it is semantic complete.

Programmatic Interpretation

The notion of satisfaction of a formula by some assignment in a model is the fundamental notion of the referential interpretation of a calculus; however, if the calculus is to represent actions, then a notion more appropriate than satisfaction is required. Success and failure of a process for a goal in a model seem to be the concepts demanded. An interpretation of a calculus in which the notion of success is the fundamental notion is called a programmatic interpretation of the calculus. Thus, in such interpretations, instead of a domain of discourse, there is a goal language in which goals are formulated. For every term that somehow represents a procedure (let us call it a program) a set of processes relative to the goal language is determined, and the germane contentive statements in this setting

refer to the co-success of processes relative to goals.

Goal Language

A goal language is a descriptive language used for the formulation of goals, and relative to which the success of processes is determined. Thus, it may be said that a goal language GL is a first order semantic system; that is, it has a formalized morphology and a formalized referential semantic framework. But GL does not have a theory proper that characterizes its non-logical constants. Hence, in the context of GL, it is proper to use the notion of a realization. (i. e., any model of the lower predicate calculus that has the appropriate structure) instead of the notion of a model.

Processes. If $R = \langle D, \delta \rangle$ is a realization of GL, V is the set of variables of GL, and $\Gamma = D^V$ is the set of assignments for R , then a process π in R can be defined as a function that maps the natural numbers into assignments; that is

$$\pi: \omega \rightarrow \Gamma \quad (16)$$

A process π is said to be bound for some set of assignments if and only if there is a natural number n such that for every natural number m if $m \geq n$ then $\pi(m)$ is a member of the set. A particular case is when the set has only one element; in this case, the process is called terminating and the assignment in the set is called the terminal assignment of the process.

Goals. If G is a set of ordered pairs of formulae of GL, and $g = \langle g_0, g_1 \rangle$ is a member of G , then g is said to be a goal. G is

called a goal set of GL. If R is a realization of GL, and g_i is a formula of GL, then the satisfaction set of g_i in R , denoted by $H(g_i)$, is defined as the set of all assignments that satisfy g_i in R . Therefore, for every goal there corresponds a pair $\langle H(g_0), H(g_1) \rangle$ of sets of assignments called the satisfaction space of the goal in the realization R .

If π is a process in some realization, and g is a goal in the goal set G , then it is said that π succeeds for g , $\text{Succ}(\pi, g)$, if and only if either $\pi(0)$ is not a member of $H(g_0)$ or π is bound for the set $H(g_1)$. It is said that π fails for g if and only if it does not succeed for g .

Two processes π and π' in some realization are said to be co-successful, $\text{Cosucc}(\pi, \pi')$, if and only if for every goal g in the goal set G

$$\text{Succ}(\pi, g) \equiv \text{Succ}(\pi', g). \quad (17)$$

As a consequence of the two-way implication in its definition, Cosucc is an equivalence relation, and as such, it generates a partition of the set of all processes in the realization. This partition remains the same or becomes finer when new goals are included in the goal set. That is, it may be the case that two processes that are co-successful for some goal set G are not co-successful when a new goal is included in G .

If the morphology of the goal language is extended, but the goal set is not modified, then such extension is irrelevant for the

effects of goal satisfaction. Therefore, it may be assumed that the goal language is as big as desired and that the goal set is what determines the relevant elements of it.

Process-Generating Agents

P is a process-generating agent for some realization of the goal language GL if and only if for some process π in that realization and for all natural number i , $\pi(i+1)$ is uniquely determined from $\pi(i)$ by P . Thus, for the generation of a process π , it is required that both the initial condition $\pi(0) = \gamma$ and the process-generating agent P be specified. This is denoted by $\pi = \langle P, \gamma \rangle$.

A function from the set of assignments into the set of assignments is an acceptable generating agent, but processes generated by it are forced to be singular. The processes generated by the iteration of the function are either non-repeating or periodic. That is, if π is a process generated by such a function, and $\pi(i) = \pi(j)$ then

$\pi(i+1) = \pi(j+1)$. Non-singular processes must be generated by some other kind of agent.

Programs. The executors of programs can also be considered to be process-generating agents. These type of agents accomplish their task by means of three mechanisms: a) a reading mechanism that maps any initial assignment in a realization of the goal language into some internal condition of the executor, determined by a suitable interpretation of a programming language; b) an executing mechanism that generates new internal conditions from the previous ones; and c) a writing mechanism that maps every internal condition into some assignment in the

realization of the goal language.

The internal conditions mentioned above are functions mapping the set of variables of the programming language into a set of values that are generally elements in a suitable interpretation of the programming language. These internal conditions are what McCarthy¹ calls state vectors. Some formalizations identify the set of variables of the programming language with the set of variables of the goal language, and the set of values with its domain of discourse. In such cases the reading and writing mechanisms become trivial; the identity functions and then the executing mechanism completely identify the process-generating agent.

The executing mechanisms for programming languages have two principal components: one fixed for the programming language (the interpreter); the other variable (the programs). Thus, if a programming language has a defined interpreter, then whenever a program is specified in that language the executing mechanism corresponding to that program is completely determined.

Combinatory Process-Generating Agents. Let us suppose that the set of variables of a combinatory logic CL and the set of variables of the goal language are identical, and denoted by V ; and suppose that T denotes the set of terms of CL, T_c denotes the set of closed terms of CL, and Σ denotes the set of addresses of CL. If $R = \langle D, \delta \rangle$ is a realization of the goal language, then the quadruple $\langle \varphi, \rho, \alpha, \delta_c \rangle$

¹McCarthy (1962), p. 24.

is a programmatic interpreter for CL in R if and only if

$\varphi : D \rightarrow T_c$; φ is called the representation function;

$\rho : T_c \rightarrow T_c$; ρ is called the execution function;

$\alpha : V \rightarrow \Sigma$; α is called the allocation function;

and δ_c is an interpretation function such that $\langle D, \delta_c \rangle$ is a referential model of CL.

For any term t in CL, a programmatic interpreter completely defines a process-generating agent for R, denoted by the quintuple $\langle \varphi, \rho, \alpha, \delta_c, t \rangle$ as follows:

- a) The elements that correspond to the assignments in R are constructions of closed terms represented in the tree of addresses of CL.
- b) The term that corresponds to the initial assignment is determined by a realization-dependent reading mechanism given by the function φ and substitution.
- c) There is a realization-independent executing mechanism that generates a sequence of terms by the iteration of the function ρ .
- d) There is a realization-dependent writing mechanism, constructed from the functions α and δ_c , that generates assignments in R corresponding to terms in CL.

Thus, if t_0 is the term corresponding to the initial assignment γ , and v_1, v_2, \dots, v_n are the variables in the non-fixed part of t , then

$$t_0 = [\varphi(\gamma(v_1)), \varphi(\gamma(v_2)), \dots, \varphi(\gamma(v_n)) / v_1, v_2, \dots, v_n] t ;$$

for all i greater than zero

$$t_i = \rho(t_{i-1}) ;$$

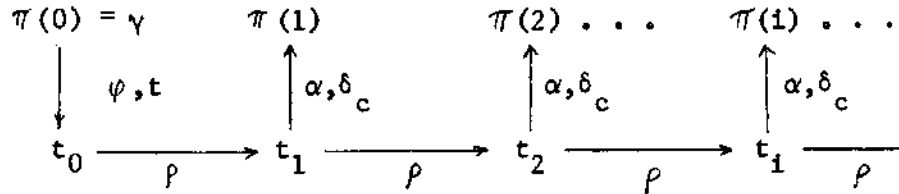
and if $\pi = \langle \varphi, \rho, \alpha, \delta_c, t, \gamma \rangle$ is the process generated by the agent $\langle \varphi, \rho, \alpha, \delta_c, t \rangle$ and initial assignment γ , then for all positive integer i and variable v

$$\pi(0) = \gamma$$

$$\pi(i)(v) = \delta_c(\alpha(v)(t_i)) \text{ if } \alpha(v)(t_i) \text{ is defined}$$

and $\pi(i)(v) = \gamma(v)$ otherwise.

These considerations can be illustrated by the following diagram



The sextuple $\langle D, \delta_G, \varphi, \rho, \alpha, \delta_c \rangle$ is a programmatic model of CL and the goal language GL if and only if $\langle D, \delta_G \rangle$ is a realization of GL and $\langle \varphi, \rho, \alpha, \delta_c \rangle$ is a programmatic interpreter for that realization.

Programmatic Equivalence

If R is a realization of GL and P_0 and P_1 are two process-generating agents defined for R , then they are programmatic equivalent, $\text{Peq}(P_0, P_1)$, if and only if for every assignment γ

$$(\langle P_0, \gamma \rangle, \langle P_1, \gamma \rangle) \quad (18)$$

Two terms t_0 and t_2 are programmatic equivalent, $\text{Progeq}(t_0, t_1)$,
 if and only if for every programmatic model $\langle D, \delta_G, \varphi, \rho, \alpha, \delta_c \rangle$
 $\text{Peq}(\langle \varphi, \rho, \alpha, \delta_c, t_0 \rangle, \langle \varphi, \rho, \alpha, \delta_c, t_1 \rangle)$. (19)

From their definitions, it is immediate that Peq , and Progeq are equivalence relations, but since they involve much semantic machinery it is desirable to have a syntactic substitute for them. That is, a necessary and sufficient condition for two terms to be programmatic equivalent is for them to be syntactic equivalent. The next theorems determine the conditions under which this can be done for a combinatory logic.

Theorem 3. For every predictor Eq in CL that satisfies the rules of reflexivity, transitivity, and symmetry, and that has an inverse of substitution, there is a non-empty goal set G for which

If $\text{Progeq}(t_0, t_1)$ then $\vdash t_0 \text{ Eq } t_1$.

Proof: Let $g = \langle g_0, g_1 \rangle$ be the only goal in G , and let $M = \langle D, \delta_G, \varphi, \rho, \alpha, \delta_c \rangle$ be a programmatic model such that

a) Every formula of CL is valid on $\langle D, \delta_c \rangle$ if and only if it is a theorem of CL. Such a model exists since CL is semantic complete in the referential sense and $\langle D, \delta_c \rangle$ is a referential model of CL by definition of programmatic model.

b) $\delta_c \subseteq \delta_G$. This is possible if we assume that the morphology of CL is included in the morphology of the goal language.

This is an assumption that can always be made without side

effects, as has been mentioned before.

- c) $\langle D, \delta_G \rangle \models g_0 [\gamma]$. This is possible since g_0 may be a tautology.
- d) g_1 is a formula of the form ' $t_g \text{ Eq } v_g$ ' where t_g is any closed term and v_g is a variable such that $\alpha(v_g)$ is the identity address.
- d) $(t \text{ Eq } \rho(t))$ for any term t .

Now let us prove that for the model M just described

If $\text{Peq}(\langle \varphi, \rho, \alpha, \delta_c, t_0 \rangle, \langle \varphi, \rho, \alpha, \delta_c, t_1 \rangle)$ then $\vdash (t_0 \text{ Eq } t_1)$

This is the same as proving that for all assignment γ

If $\text{Cosucc}(\langle \varphi, \rho, \alpha, \delta_c, t_0, \gamma \rangle, \langle \varphi, \rho, \alpha, \delta_c, t_1, \gamma \rangle)$ then $\vdash (t_0 \text{ Eq } t_1)$

1. $\text{Succ}(\langle \varphi, \rho, \alpha, \delta_c, t_0, \gamma \rangle, g)$ iff
2. $\langle D, \delta_G \rangle \models g_1 [\pi(m)]$ where m is any natural number greater than some natural number n , and

$$\pi(m)(v) = \delta_c(\alpha(v)(\rho^m([\varphi(\gamma(v_1)), \dots, \varphi(\gamma(v_k))/v_1, \dots, v_k]t_0)))$$
 or $\pi(m)(v) = \gamma(v)$ from condition c
3. $\langle t_g [\pi(m)], v_g [\pi(m)] \rangle \in \delta_G (\text{Eq})$ from 2 and definition
4. $\langle \delta_c(t_g), \delta_c(\rho^m([\varphi(\gamma(v_1))/v_1]t_0)) \rangle \in \delta_c (\text{Eq})$ from conditions b, d
5. $\vdash (t_g \text{ Eq } \rho^m([\varphi(\gamma(v_1))/v_1]t_0))$ from 4, and condition a
6. $\vdash (t_g \text{ Eq } [\varphi(\gamma(v_1))/v_1]t_0)$ from 5, condition e, & TRN
7. $\vdash (t_g \text{ Eq } [\varphi(\gamma(v_1))/v_1]t_0) \equiv \text{Succ}(\langle \varphi, \rho, \alpha, \delta_c, t_0, \gamma \rangle, g)$ from 1 - 6

8. $\vdash (t_g \text{ Eq } [\varphi(\gamma(v_i))/v_i]t_i) \equiv \text{Succ } (<\varphi, \rho, \alpha, \delta_c, t_i, \gamma>, g)$
from 7
9. $\vdash (t_g \text{ Eq } [\varphi(\gamma(v_i))/v_i]t_0) \equiv \vdash (t_g \text{ Eq } [\varphi(\gamma(v_i))/v_i]t_1)$
from 8, 7
10. $\vdash [\varphi(\gamma(v_i))/v_i]t_0 \text{ Eq } [\varphi(\gamma(v_i))/v_i]t_1$ from 9, TRN, SYM
11. $\vdash t_0 \text{ Eq } t_1$ since Eq has an inverse of substitution and
 φ has not been restricted.

Since $(M)(\text{Peq}(P_0, P_1))$ implies that $(\exists M)(\text{Peq}(P_0, P_1))$ then
 $\text{Progeq}(t_0, t_1)$ implies $\vdash t_0 \text{ Eq } t_1$.

Theorem 4. For every predicate Eq in CL that satisfies the
rules of reflexivity, transitivity, and symmetry and that has an
inverse of substitution, if the set of programmatic models is
restricted to those models that satisfy

- a) $\delta_c \subseteq \delta_G$
- b) $\vdash t \text{ Eq } \rho(t)$ for every term t
- c) $\alpha(v_g)$ is the identity address for some v_g

then there is a non-empty goal set G for which

$$\text{Progeq}(t_0, t_1) \equiv \vdash t_0 \text{ Eq } t_1$$

Proof: Let $g = <g_0, g_1>$ be the only goal in G , then the
proof of the theorem is in the inverse order of the proof of the
previous theorem, with the difference that in that theorem conditions
a), b), c) are satisfied by one model only and in this theorem it is
necessary to force them to be satisfied by all models.

Theorem 5. If for every programmatic model of CL the execution

function p is restricted to satisfy the condition that $t_0 \text{ Eq } t_1$ implies the existence of a natural number n such that

$$p^n(t_0) = p^n(t_1),$$

then for every goal set G , Eq is a sufficient condition for programmatic equivalence.

Proof: By construction, for every sequence of terms generated by the iteration of the execution function, there corresponds a process in every model determined by the reading and writing mechanisms of the programmatic interpreter of the model. If both sequences of terms are such that after some natural number they have the same terms, then the corresponding processes have the same assignments after the same natural number, and therefore they are bound for the same sets of assignments. Hence, they co-succeed for any goal in the model.

By hypothesis, and based on the rule of substitution and the conclusion of the preceding paragraph, it may be further concluded that Eq is a sufficient condition for programmatic equivalence.

CHAPTER IV

CONCLUDING REMARKS

Conclusions

Completeness of Convertibility

The most important conclusion of this investigation is what may be called the programmatic completeness of convertibility. From theorem 1, it is known that ' \Leftrightarrow ' has an inverse of substitution; it therefore has all the qualifications needed by the predicator Eq in theorem 3. There is thus a goal set for which convertibility is a necessary condition for programmatic equivalence; furthermore, for every goal set which includes that minimal goal set, convertibility is also a necessary condition for programmatic equivalence.

If the restriction indicated in the formulation of theorem 5 is made, then convertibility is a necessary and sufficient condition for programmatic equivalence for any goal set that includes a goal of the form:

$$\langle g_0, t \Leftrightarrow v \rangle \quad (20)$$

where g_0 is a tautology, t is any closed term, and v is any variable. This goal may be called a kernel goal of convertibility.

The importance of this result is that mechanical proofs of convertibility can be used for the proof of programmatic equivalence of programs in the same, or even different, programming languages.

In the way that programmatic models were constructed for combinatory logics, programmatic models can be constructed for programming languages, and programmatic equivalence can be formulated in those models as it was formulated for the models of the combinatory logics. Moreover, if composed programmatic models for two programming languages are constructed, then the problem of compiler correctness for these languages can be formulated, studied, and hopefully, proved.

Thus, if the correctness of a compilation from some programming language to a combinatory logic is proved, then the proof of the programmatic equivalence of programs in that programming language can be reduced to the proof of the programmatic equivalence of their combinatory compilations, and this--as a consequence of the main result of this dissertation--corresponds to the proof of convertibility of such compilations.

A warning should be made concerning the inclusion of a kernel goal of convertibility in the goal set with respect to the programmatic equivalence of programs in a programming language, since the possibility exists that such an inclusion may produce an undesirable effect in some language. This may especially be true in composed models where combinatory compilers have been studied.

Completeness of Extensional Equivalence

The second important conclusion of this investigation is a consequence of theorems 1 and 4. From theorem 1 ' \approx ' has an inverse of substitution, and from theorem 4 ' \approx ' is a necessary and sufficient condition for programmatic equivalence for some non-empty goal set G,

provided the appropriate restrictions are made on the set of models.

The attractiveness of this result derives from the fact that extensional equivalence is a relation larger than convertibility and that, under some conditions in which the introduction of the kernel goal of convertibility might produce undesirable results, extensional equivalence may be an appropriate alternative.

The deficiency of the result is that it requires a strong restriction on the set of goals and on the set of models; thus, the range of applicability of the result is not as broad as could be desired.

Future Research

The research here reported covers only one fundamental aspect of a wider program of investigation oriented towards the development of a logic of programming. The next steps to be taken in this direction are:

1. Extend the notion of programmatic models, as defined for combinatory logics, to programming languages. Most of the work done in the area of semantics of programming languages accepts the notion of an interpreter transforming a state vector as the meaning of a program. Within the context of this dissertation, this notion is true when the programmatic model is a free model; that is, when the domain of interpretation of the model is the same as the set of values of the variables in the state vector.
2. Develop a semantic framework for the programmatic interpretation of compiling--in particular, compiling from a programming

language to a combinatory logic. In such a framework, define the notion of compiler correctness. Prove the correctness of some of the available translators. And develop a language-independent combinatory compiler that generates correct combinatory terms from programs in some programming language, given the program, the syntax of the language, and the programmatic semantics as defined in 1.

3. Study the effect of the restrictions in theorems 3 - 5 in selected programming languages, and examine the relationship between classical syntactical equivalence relations on programming languages and combinatory equivalence relations. Investigate possible extensions of the equivalence relations discussed in this dissertation.
4. Compare the notion of goal as a pair of goal formulae in the goal language with the notion of goal in cybernetics. Study the restrictions generated by considering goal as it is viewed in this dissertation. (A possible extension of this notion would be to consider goals as sequences of goal formulae.)
5. Develop algorithms for the proofs of convertibility and extensional equivalence. However, it should not be forgotten that this problem has been proven undecidable for the most general case.

BIBLIOGRAPHY

- Barendregt, H. P., (1971), "Some Extensional Term Models for Combinatory Logics and λ -Calculi" Doctoral Thesis, U. Utrecht.
- Bell, J. L., and Slomson, A. B., (1969), Models and Ultraproducts; an Introduction. North-Holland, Amsterdam.
- Blum, E. K., (1969), "Towards a Theory of Semantics and Compilers for Programming Languages" J. Comp. and Syst. Sci., 3:248-275.
- Böhm, C., (1966), "The CUCH as a Formal and Descriptive Language." Proceeding of the IFIP Working Conference on Formal Language Description Languages (Ed. Steel T. B.), 179-197.
- Böhm, C. and Gross, W., (1966), "Introduction to the CUCH." in Automata Theory (Ed. Caianiello, E. R.), 35-65.
- Braffort, P. and Hirschberg, D. (Eds.) (1963), Computer Programming and Formal Systems, North-Holland, Amsterdam.
- Burstall, R. M., and Landin, P. J., (1969), "Programs and their Proofs: an Algebraic Approach" Machine Intelligence, 4:17-43.
- Caianiello, E. R. (Ed.), (1966), Automata Theory, Academic Press, N. Y.
- Caracciolo Di Forino, A., (1965), "Linguistic Problem in Programming Theory" in Information Processing 1965 (Ed. Kalenich, W. A.), 223-228.
- Curry, H. B. and Feys, R., (1958), Combinatory Logic, Vol I. North-Holland, Amsterdam.
- Curry, H. B.; Hindley, R. J. and Seldin, J. P., (1972), Combinatory Logic, Vol. II North-Holland, Amsterdam.
- De Bakker, J. W., (1970), "Semantics of Programming Languages." Advances in Information Systems Science, 2:173-227.
- De Millo, R. A., (1972), "Formal Semantics and the Logical Structure of Programming Languages." Doctoral Thesis, School of Info. and Comp. Science. Georgia Institute of Technology.
- Droege, S., (1971), "On the Practicality of Manna's Method of Verifying the Termination and Correctness of Programs." Department of Computer Science, Technical Report #11, Rutgers University.

- Elspas, B. et al., (1972), "An Assessment of Techniques for Proving Program Correctness." *Comp. Surveys*, 4:97-147.
- Floyd, R. W., (1967), "Assigning Meanings to Programs." in Mathematical Aspects of Computer Science (Ed. Schwartz, J. T.), 19-32.
- Fox, J. (Ed.), (1971) *Computers and Automata*. Polytechnic Press, Brooklyn, N. Y.
- Iverson, K. E., (1962), *A Programming Language*, Wiley, New York.
- Kalenich, W. A. (Ed.), (1965), *Information Processing 1965* Spartan Books, Washington, D. C.
- Landin, P. J. (1964), "The Mechanical Evaluation of Expressions." *Computer J.*, 6:308-320.
- Landin, P. J., (1965), "A Correspondence Between ALGOL 60 and Church's λ -notation." *Comm. ACM*, 8:89-101 and 8:158-167.
- Landin, P. J., (1966), "A Formal Description of ALGOL 60" in Formal Language Description Languages (Steel, T. B. Ed.) 266-294.
- Lucas, P. and Walk, K., (1970), "On the Formal Description of PL/I." *Annual Review in Automatic Programming*, 6:105-181.
- Luckham, D. C., Park, D.M.R., and Paterson, M. S., (1970), "On Formalized Computer Programs." *J. Comp. and Syst. Sci.*, 4:220-249.
- McCarthy, J., (1962), "Towards a Mathematical Science of Computation." in Information Processing 1962 (Popplewell, C. M. Ed.) 21-28.
- McCarthy, J., (1963), "A Basis for a Mathematical Theory of Computation." in Computer Programming and Formal Systems (Eds. Braffort P. and Hirschberg D), 33-70.
- McCarthy, J., (1965), "Problems in the Theory of Computation" in Information Processing 1965 (kalenich, W. A. Ed.) 219-222.
- McCarthy, J., (1966), "A Formal Description of a Subset of Algol." in Formal Language Description Languages (Steel, T. B. Ed.) 1-12.
- McCarthy, J., and Painter, J., (1967), "Correctness of a Compiler for Arithmetic Expressions" (Ed. Schwartz, J. T.), 33-41.
- Manna, Z., (1969), "The Correctness of Programs" *J. Comp. and Syst. Sci.*, 3:119-127.
- Manna, Z. and McCarthy, J., (1970) "Properties of Programs and Partial Function Logic" *Machine Intelligence*, 5:27-37.

- Morris, J. H., (1968), "Lambda-Calculus Models of Programming Languages" Doctoral Thesis, School of Management, Massachusetts Institute of Technology.
- Naur, P. (Ed.), (1963), "Revised Report on the Algorithmic Language ALGOL 60." Comm. ACM, 6:1-17.
- Orgass, R. J., (1967), "A Mathematical Theory of Computing Machine Structure and Programming." Doctoral Thesis, Yale University.
- Orgass, R. J. (1970), "Some Results Concerning Proofs of Statements About Programs." J. Comp. and Syst. Sci., 4:74-88.
- Orgass, R. J. and Fitch, F. B., (1969), "A Theory of Computing Machines." Studium Generale, 22:83-104.
- Orgass, R. J. and Fitch, F. B., (1969), "A Theory of Programming Languages." Studium Generale, 22:113-136.
- Petznick, G. W., (1970), "Combinatory Programming" Doctoral Thesis, Department of Computer Science, University of Wisconsin.
- Popplewell, C. N. (Ed.), (1962), Information Processing 1962, North-Holland, Amsterdam.
- Robinson, A., (1965), Introduction to Model Theory and to the Metamathematics of Algebra. (2nd Ed.) North-Holland, Amsterdam.
- Rustin, R. (Ed.), (1972), Formal Semantics of Programming Languages, Prentice-Hall, Englewood Cliffs, N. J.
- Schwartz, J. T. (Ed.), (1967), Mathematical Aspects of Computer Sciences, Amer. Math. Soc. Vol. 19, Providence, R. I.
- Scott, D. and Strachey, C., (1971), "Toward a Mathematical Semantics for Computer Languages." in Computers and Automata (Fox, J. Ed.) 19-46.
- Steel, T. B. (Ed.), (1966), Formal Language Description Languages for Computer Programming. North-Holland, Amsterdam.
- Strachey, C., (1966), "Towards a Formal Semantics" in Formal Language Description Languages (Steel, T. B. Ed.) 198-220.
- Van Fraassen, B. C., (1971), Formal Semantics and Logic, MacMillan, N. Y.
- Venturini-Zilli, M., (1965), " λ -K-Formulae for Vector Operators." ICC Bulletin, 4:157-174.
- Wadsworth, C., (1971), "Semantics and Pragmatics of the λ -Calculus" Doctoral Thesis, Oxford University.

Wegner, P., (1972a), "Programming Language Semantics," in Formal Semantics of Programming Languages (Rustin, R. Ed.), 149-248.

Wegner, P., (1972b), "The Vienna Definition Language." Comp. Surveys 4:5-63.

VITA

Jorge Baralt-Torrijos was born in San Cristóbal, Venezuela, on May 4, 1943. He received the degree of Civil Engineer from the Universidad Central de Venezuela in 1966. From 1968 to 1970 he attended the Georgia Institute of Technology, where he earned the degree of Master of Science in Information and Computer Science.

He was employed by Cia. Shell de Venezuela, Ltd. from 1966 to 1972 as a system programmer and a system analyst. Since 1972 he has been in charge of the Coordination of the career on Computer Science at the Universidad Simón Bolívar. He has held part-time positions teaching at the Universidad Central de Venezuela and at the Instituto de Estudios Superiores de Administración.

Mr. Baralt-Torrijos has three publications, and has read two papers before regional conferences. He has been a member of the Association for Computing Machinery since 1968, of the American Society for Information Science since 1968, and of the Asociación Venezolana de Ingeniería de Computación Electrónica (AVICE) since 1966. He was president of AVICE for the period 1972-1973.