IMPLEMENTATION AND APPLICATIONS OF SCOTT'S LOGIC FOR COMPUTABLE FUNCTIONS*

by

Robin Milner
Stanford University
Stanford, California

Introduction

The basis for this paper is a logic designed by Dana Scott [1] in 1969 for formalizing arguments about computable functions of higher type. This logic uses typed combinators, and we give a more or less direct translation into typed λ-calculus, which is an easier formalism to use, though not so easy for the metatheory because of the presence of bound variables. We then describe, by example only, a proof-checker program which has been implemented for this logic; the program is fully described in [2]. We relate the induction rule which is central to the logic to two more familiar rules - Recursion Induction and Structural Induction - showing that the former is a theorem of the logic, and that for recursively defined structures the latter is a derived rule of the logic. Finally we show how the syntax and semantics of a simple programming language may be described completely in the logic, and we give an example of a theorem which relates syntactic and semantic properties of programs and which can be stated and proved within the logic.

PURE LCF

We reserve the name LCF (Logic for Computable Functions) for the computer implementation, and give the name PURE LCF to the translation of Scott's logic into λ-calculus terms. LCF is a pollution of PURE LCF only in the sense that it attempts to make the process of generating proofs convenient.

Types

There are denumerably many types, and the type of each term is unambiguously determined. At bottom are the types tr and ind (truth-values and individuals). Further, if $\beta 1$ and $\beta 2$ are types then $(\beta 1 \rightarrow \beta 2)$ is a type (the type of functions whose domain and range have types $\beta 1, \beta 2$). We often omit parentheses and write $\beta 1 \rightarrow \beta 2$. Also, adopting the convention that "$\rightarrow$" associates to the right, we write $\beta 1 \rightarrow \beta 2 \rightarrow \beta 3$ for $(\beta 1 \rightarrow (\beta 2 \rightarrow \beta 3))$. We write

$$t : \beta$$

to mean that term t has type β. Throughout, we use $\beta, \beta 1, \beta 2, ..$ as metavariables for types.

*The research reported here was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183).

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Terms

(Metavariables s,t,s1,t1,..)

The following are terms:

(i) Identifiers - i.e. sequences of letters and digits.
(Metavariables x,y, x1, y1,..). We assume that the type of each identifier is determined.

(ii) Applications - $s(t) : \beta 2$, where $s : \beta 1 \rightarrow \beta 2$ and $t : \beta 1$.

(iii) Conditionals - $(s \rightarrow t1, t2) : \beta$, where $s : tr$ and $t1, t2 : \beta$.

(iv) λ-expressions - $[\lambda x.s] : \beta 1 \rightarrow \beta 2$, where $x : \beta 1$ and $s : \beta 2$.

(v) α-expressions - $[\alpha x.s] : \beta$, where $x, s : \beta$.

Semantics of Terms

Terms of type tr, ind denote truth-values, individuals. A term of type $\beta 1 \rightarrow \beta 2$ denotes a monotonic continuous partial function whose range, domain have types $\beta 2$, $\beta 1$. The identifiers TT,FF:tr denote "true", "false" - i.e. they are constants. The only other constant is UU which denotes the undefined object at any type - in particular the undefined truth-value. Strictly there are denumerably many UU's - one at each type- and they should be tagged with their types; we leave the tags off, relying on the context to determine the tag for each occurrence of UU.

The interpretation of (i)-(iv) is obvious. The intended interpretation of $[\alpha x.s]$ is the minimal fixed-point of the function denoted by $[\lambda x.s]$. (α is the equivalent of "LABEL" in LISP). Thus

$$[\alpha f.[\lambda x.(p(x) \rightarrow f(a(x)), b)]]$$

denotes the function defined recursively by

$$f(x) \Leftarrow \text{if } p(x) \text{ then } f(a(x)) \text{ else } b.$$

For a justification and discussion of the semantics, see [1].

Atomic wffs, or awffs

If s,t are terms then

$$s \subset t$$

is an awff. It is interpreted as "the object denoted by t is at least as well defined as, and consistent with, that denoted by s".

Wffs

(Metavariables P,Q,P1,Q1,..)

Wffs are sets of zero or more awffs, written as lists separated by commas. They are interpreted as conjunctions. We use

$$s \equiv t$$

to abbreviate $s \subset t$, $t \subset s$.

Sentences

Sentences are written

$$P \supset Q$$

where P,Q are wffs, and are interpreted as implications. If P is empty, we write

$$\supset Q$$

Proofs

A proof is a sequence of sentences, each being derived from zero or more preceding sentences by a rule of inference.

Rules of Inference

We write the hypotheses above, and the conclusion below, a horizontal line. We write

$$P\{s/x\} \quad \text{or} \quad t\{s/x\}$$

to mean the result of substituting term s for all free occurrences of identifier x in P or t, after first systematically changing bound identifiers in P or t so that no identifier free in s becomes bound by the substitution. Only λ α bind identifiers.

***** $\supset$ - RULES *****

INCL $\quad \dfrac{}{P \supset Q}$ (Q a subset of P)

CONJ $\quad \dfrac{P1 \supset Q1 \qquad P2 \supset Q2}{P1 \cup P2 \supset Q1 \cup Q2}$

CUT $\quad \dfrac{P1 \supset P2 \qquad P2 \supset P3}{P1 \supset P3}$

***** $\subset$ - RULES *****

APPL $\quad \dfrac{P \supset s1 \subset s2}{P \supset t(s1) \subset t(s2)}$

REFL $\quad \dfrac{}{P \supset s \subset s}$

TRANS $\quad \dfrac{P \supset s1 \subset s2 \qquad P \supset s2 \subset s3}{P \supset s1 \subset s3}$

***** UU - RULES *****

MIN1 $\quad \dfrac{}{\supset UU \subset s}$

MIN2 $\quad \dfrac{}{\supset UU(s) \subset UU}$

***** CONDITIONAL RULES *****

CONDT $\quad \dfrac{}{\supset TT \rightarrow s,t \equiv s}$

CONDU $\quad \dfrac{}{\supset UU \rightarrow s,t \equiv UU}$

CONDF $\quad \dfrac{}{\supset FF \rightarrow s,t \equiv t}$

***** λ - RULES *****

ABSTR $\quad \dfrac{P \supset s \subset t}{P \supset [\lambda x.s] \subset [\lambda x.t]}$ (x not free in P)

CONV $\quad \dfrac{}{\supset [\lambda x.s](t) \equiv s\{t/x\}}$

FUNC $\quad \dfrac{}{\supset [\lambda x.y(x)] \equiv y}$

***** TRUTH-RULE *****

CASES $\quad \dfrac{P, s\equiv TT \supset Q \quad P, s\equiv UU \supset Q \quad P, s\equiv FF \supset Q}{P \supset Q}$

***** α RULES *****

FIXP $\quad \dfrac{}{\supset [\alpha x.s] \equiv s\{[\alpha x.s]/x\}}$

INDUCT $\quad \dfrac{P \supset Q\{UU/x\} \qquad P, Q \supset Q\{t/x\}}{P \supset Q\{[\alpha x.t]/x\}}$ (x not free in P)

LCF On the Computer

A proof-generator program, written in MLISP, due to Enea and Smith [3,4], is running on the PDP-10 at the Artificial Intelligence Project, Stanford. (Actually we used MLISP2, an extendible language consisting of MLISP + language definition facility.) The program does not yet aim at automatic theorem-proving. Each rule of inference is available as a command, and there is also an element of goal structure which gives a flavor of Natural Deduction. A simple example will give an idea of how to generate proofs on the computer. We can kill two birds with one stone by showing how to prove a version of Recursion Induction - see McCarthy [5] - thus relating the latter to the Scott Induction Rule. See also de Bakker and Scott [6] for essentially the same proof. One may state Recursion Induction as follows: "if f is a recursively defined function and g, another function, satisfies f's defining equation then g extends f". IN PURE LCF we can state this as

$$g \equiv fun(g) \supset [\alpha f.fun(f)] \subset g$$

but the version we will prove is

$$f \equiv [\alpha f.fun(f)],\ g \equiv fun(g) \supset f \subset g$$

Below is the conversation between user and computer. The Roman numerals at the right refer to the notes which follow. The user's commands are always preceded by *****; everything else is the computer's reply.

```
*****ASSUME f≡[αf.fun(f)], g≡fun(g);
1   f≡[αf.fun(f)]  (1)                                   (i)
2   g≡fun(g)  (2)                                        (i)

*****GOAL f ⊂ g;                                        (ii)
NEWGOAL #1 f ⊂ g                                        (ii)

*****TRY 1 INDUCT 1;                                    (ii)
NEWGOAL #1#1 UU ⊂ g                                     (ii)
NEWGOAL #1#2 fun(f1)⊂g  ASSUME f1⊂g                     (ii)
```

```
*****TRY 1;                                                    (iii)
*****MIN1 g;                                                   (iii)
3  UU ⊂ g                                                      (iii)
*****QED;                                                      (iii)
GOAL #1#1 PROVED.  BACK UP TO GOAL #1                          (iii)
REMAINING SUBGOALS:                                            (iii)
#1#2  fun(f1) ⊂ g  ASSUME f1 ⊂ g                               (iii)

*****TRY 2;                                                    (iv)
4  f1⊂g  (4)                                                   (iv)
*****APPL  fun,4;                                              (iv)
5  fun(f1)⊂fun(g)  (4)                                         (iv)
*****SYM 2;                                                    (iv)
6  fun(g)≡g  (2)                                               (iv)
*****TRANS 5,6;                                                (iv)
7  fun(f1)⊂g  (2 4)                                            (iv)
*****QED;                                                      (iv)
GOAL #1#2 PROVED.  BACK UP TO GOAL #1.  NO MORE
   SUBGOALS                                                    (v)
8  f⊂g  (2 1)                                                  (v)
GOAL #1 PROVED.  BACK UP TO TOP LEVEL.  NO MORE
   SUBGOALS                                                    (v)
*****SHOW PROOF;                                               (vi)
```

PROOF

```
1    f ≡ [αf.fun(f)]  (1)  ----- ASSUME.
2    g ≡ fun(g)   (2)  -----ASSUME.

TRY #1   f ⊂ g                INDUCT 1.

   TRY #1#1        UU ⊂ g
   3   UU ⊂ g      ------ MIN1 g.

   TRY #1#2        fun(f1)  ⊂ g : ASSUME f1 ⊂ g
   4     f1 ⊂ g    (4) ------ ASSUME.
   5     fun(f1) ⊂ fun(g)  (4) ------ APPL fun 4.
   6     fun(g) ≡ g  (2)  ------ SYM 2.
   7     fun(f1) ⊂ g  (4 2)  ------ TRANS 5 6.

8   f  ⊂  g  (2 1)  ------- INDUCT 3 7.
```

Notes

(i) The machine generates steps of a proof under user control. Each step is

n P S

where n is a stepnumber, P a wff and S a list of stepnumbers. Such a step represents the sentence Q ⊃ P where Q is the union of wffs referenced by S. In this case steps 1 and 2, generated by the ASSUME command, are just instances of P ⊃ P - a case of the INCL rule.

(ii) The user states a goal; the machine gives it a goal index-number, #1, and the user states that he wants to prove it by induction using step 1 - the recursive definition of f. The machine specifies as new subgoals #1#1, #1#2 the hypotheses of the appropriate instance of the induction rule.

(iii) The user attacks the first subgoal (TRY 1). He gives no tactic this time, but proves it himself by using an appropriate instance (specified by the parameter g) of the minimality rule MIN1. When he achieves the subgoal he says QED; the machine agrees, back up the goaltree, and tells him what subgoals remain.

(iv) The user attacks subgoal 2, again without tactic. This time there is an assumption to make (f1 ⊂ g), and the machine makes it for him. The user proves the goal in three steps. Step 6 is by the symmetry rule for ≡, which saves him unabbreviating s ≡ t to s ⊂ t, t ⊂ s.

(v) The machine backs right up the goaltree this time, recording the proof of the main goal as step 8.

(vi) The user asks for the whole proof to be displayed. Note that the goal structure is also shown. He can if he wishes to preserve the proof on file.

Let us give another example, this time a theorem in a real computation domain - lists. We omit the proof, but give the example merely to show

(i) how a certain computation domain can be axiomatized (though we do not give all the axioms for lists);

(ii) how various abbreviations allow quite a natural expression of the theorem.

The theorem states the associativity ap, the append function for lists. It may be written

$$\begin{aligned} &\text{null}\ \ UU \equiv UU \\ &\forall x\ y.\ \text{hd}(\text{cons}\ x\ y) \equiv x, \\ &\forall x\ y.\ \text{tl}(\text{cons}\ x\ y) \equiv y, \\ &\forall x\ y.\ \text{null}(\text{cons}\ x\ y) \equiv FF, \end{aligned}$$

$$ap \equiv \alpha f.\ \lambda x\ y.\ \text{null}\ x \rightarrow y,\ \text{cons}(\text{hd}\ x,\ f(\text{tl}\ x,\ y)) \supset\ ap(x,\ ap\ y\ z) \equiv ap(ap\ x\ y,\ z)$$

We have used the following sorts of abbreviation (they can of course be given a precise syntax):

hd x	for hd(x)
cons(x,y) or cons x y	for cons(x)(y)
[λx y ... .t]	for [λx. [λy. ... .t ...]]
λx y ... .t	for [λx y ... .t]
αx. t	for [αx. t]
∀x. s ⊂ t	for λx.s ⊂ λx.t
∀x y ... .P	for ∀x. ∀y. ... .P

For practical use - in particular for the example in the next section - we need further machinery. First, another abbreviation, which expresses conveniently the relativization of a wff to a truth-valued term. We write

s :: t1 ⊂ t2 for s → t1,UU ⊂ s → t2,UU

and similarly with ≡ in place of ⊂.
For example, the axiom which states that nil is the only null object reads

$$\forall x.\ \text{null}\ x :: x \equiv nil.$$

Second, we need the propositional connectives (the first two as infixes)

$$\wedge, \vee\ :\ tr \rightarrow tr \rightarrow tr$$
$$\neg\ :\ tr \rightarrow tr$$

and for these we can give the following axioms:

$$\neg \equiv \lambda x.\ x \rightarrow FF, TT,\ \wedge \equiv \lambda x\ y.\ \neg(\neg\ x \vee \neg\ y),$$
$$\forall x\ y.\ x \vee y \equiv y \vee x\ ,\ \forall x.\ x \vee FF \equiv x\ ,\ \forall x.\ x \vee TT \equiv TT$$

corresponding to the truth-tables.

$\neg$	
TT	FF
UU	UU
FF	TT

$\wedge$	TT	UU	FF
TT	TT	UU	FF
UU	UU	UU	FF
FF	FF	FF	FF

$\vee$	TT	UU	FF
TT	TT	TT	TT
UU	TT	UU	UU
FF	TT	UU	FF

Third, we need a monotonic equality predicate over individuals

$$= : \text{ind} \to \text{ind} \to \text{tr}$$

which is different from $\equiv$, since the latter is not monotonic and therefore not admissible within terms. The following two axioms for = appear to be sufficient, and are in a sense categorical:

$$\forall x\ y.\ x=y :: x \equiv y$$
$$\forall x\ y.\ x=x \wedge y=y \equiv x=y \wedge y=x$$

We can use $x=x \equiv TT$ to characterize the definedness or determinedness of x; there is little difference between recognizing an individual as determined, and recognizing its self-identity! The term x=x seems sufficiently important to define

$$\partial \equiv \lambda x.\ x=x$$

which is the monotonic version of McCarthy's predicate "*" [7]. For example, we can express totality of a function f : ind $\to$ ind by

$$\forall x.\ \partial(x) \subset \partial(f(x))$$

However, we will not discuss this further.

Fourth, and perhaps most important, we need to be able to do structural induction naturally within the logic; actually the machinery for this is already present with Scott's Induction rule. Let us here consider list structures as an example.

If P is a wff with free variable x representing the required property of lists, we wish to prove the relativized assertion

$$\forall x.\ \text{islist } x :: P$$

where

islist $\equiv \alpha f.\text{listfun}(f)$,
listfun $\equiv \lambda f\ x.$ null x $\to$ TT, f(hd x) $\to$ f(tl x), UU
.....(L)

(actually islist characteristics finite lists built entirely from the null element). Now the following instance of the induction rule represents a (course-of values) structural induction on lists:

$$\frac{Q \supset \forall x.\ UU(x)::P \qquad Q, \forall x.f(x)::P \supset \forall x.\text{listfun}(f)(x)::P}{Q \supset \forall x.\text{islist}(x)::P}$$

where $L \subseteq Q$. The first hypothesis - the induction base - is trivially satisfied (by unabbreviating ::), and the second - the induction step - may be phrased "if P is true for all x in an arbitrary set f, then it is true for all x in listfun(f) - i.e. for all lists which are either null or have their head and tail in f".

One may also show that the following somewhat more easy-to-use rule is a derived rule of the logic:

$$\frac{Q \supset \text{null } x :: P \qquad Q,\ P\{\text{hd}(x)/x\},\ P\{\text{tl}(x)/x\} \supset P}{Q,\ L \supset \text{islist } x :: P}$$

where again $L \subseteq Q$. However, the present implementation of LCF has no facility for deriving rules.

Proofs About Programs

We show by an example how formal syntax and formal semantics of a programming language may be expressed in the logic in such a way that theorems relating semantic and syntactic properties may be stated and proved also within the logic. The example we take is as follows, informally. Suppose that a programming language consists of only assignments, conditional statements, while statements and compound statements. Suppose that p is a program, e an expression and n a program variable such that neither n nor any variable in e occurs as the destination of an assignment in p. The theorem we wish to formalize and prove is that under the above restriction the following are equivalent programs:

(i) The assignment n:=e followed by the program p;
(ii) The program p with n everywhere replaced by e, followed by the assignment n:=e.

First, we need many kinds of individuals, some syntactic and some semantic. The syntactic ones are names (i.e. variables), operators, expressions, assignments, conditional statements, while statements and compound statements; the semantic ones are just the values which variables may take at run-time. Of course, as far as the logic is concerned, all these individuals are part of the semantics, and will be denoted by terms of type ind. We denote for example a function from names to values by a term of type ind $\to$ ind, and the axioms may not characterize the behavior of such a function on arguments which are not names.

We can represent the abstract syntax of programs - see McCarthy [9] -

(i) Assuming for names and operators the characteristic predicates

isname, isop : ind $\to$ tr

(ii) Associating with each other kind of syntactic individual a constructor function, some selector functions and a characteristic predicate. For example, for assignments:

mkass : names $\to$ expressions $\to$ assignments
lhsof : assignments $\to$ names
rhsof : assignments $\to$ expressions
isass : ind $\to$ tr

with axioms

$\forall n\ e.$ lhsof(mkass n e) $\equiv$ n,
$\forall n\ e.$ rhsof(mkass n e) $\equiv$ e,
$\forall n\ e.$ isass(mkass n e) $\equiv$ TT,
$\forall p.$ isass p :: mkass(lhsof p, rhsof p) $\equiv$ p

We also need axioms expressing the disjointness of the characteristic predicates, such as

$\forall p.$ isass p :: isname p $\vee$ isexpr p $\vee$ iscond $\vee$ iswhile p $\vee$ iscmpnd p $\equiv$ FF

(iii) Characterizing the classes of well-formed expressions and programs by recursively defined predicates:

iswfexpr $\equiv \alpha f.\ \lambda e.$ isname e $\to$ TT,
isexpr e $\to$ isop(opof e)$\wedge$f(arglof e)$\wedge$
f(arg2of e), UU
iswfprog $\equiv \alpha f.\ \lambda p.$

```
isass p → isname(lhsof p) ∧ iswfexpr(rhsof
   p),
iscond p → iswfexpr(condof p)∧f(lbodyof p)∧
    f(rbodyof p),
iswhile p → iswfexpr(testof p)∧f(bodyof p),
iscmpnd p → f(firstof p)∧f(secondof p),UU
```

where iscond, condof, lbodyof, rbodyof are the predicate and selectors associated with conditionals, and similarly for the other disjuncts.

We can now define recursively the predicate

ischanged : expressions → programs → tr

so that for well-formed expression e, program p, ischanged(e,p) is true iff some name in e occurs as the lhs of some assignment in p. We can also define recursively the infixed substitution combinator

/ : expressions → names → programs → programs

so that the function

e/n : programs → programs

substitutes e for n everywhere in a program. We omit the definitions of these two functions.

Now consider the semantics of programs. We suppose given a function

MOP : operators → values → values → values

which associates with each operator a binary function over values. We take a state to be a function from names to values, and a state function to be a function from states to states:

```
states = (names → values)
statefunctions = (states → states)
```

and we define the semantic functions of expressions and programs

```
MEXPR : expressions → states → values
MPROG : programs → statefunctions
```

recursively as follows:

```
MEXPR ≡ αM. λe. λs.
        isname e → s e,
        isexpr e → MOP(opof e)(M(arglof e,s),
          M(arg2of e,s)),UU
MPROG ≡ αM. λp.
        isass p → (lhsof p ← rhsof p),
        iscond p →COND(MEXPR(condof p),
          M(lbodyof p),M(rbodyof p)),
        iswhile p → WHILE(MEXPR(testof p),
          M(bodyof p)),
        iscmpnd p → M(firstof p)⊗M(secondof p),
        UU
```

where

```
← : names → expressions → statefunctions
COND : (states → tr) → statefunctions →
       statefunctions → statefunctions
WHILE : (states → tr) → statefunctions →
        statefunctions
⊗ : statefunctions → statefunctions →
    statefunctions
```

are defined by

```
← ≡ λn e s m. m=n → MEXPR(e,s), s(m)
COND ≡ λg f g x. q(x) → f(x),g(x)
WHILE ≡ λg f. αg. COND(g, f⊗g, λs.s)
⊗ ≡ λf g s. g(f(s))
```

We are now ready to formulate the theorem stated informally at the start of this Section. Let Q denote the conjunction of all the following wffs:

(i) Those which axiomatize =, ∧, ∨, ¬;
(ii) Those which axiomatize the abstract syntax;
(iii) Those which define syntactic predicates and functions;
(iv) Those which define semantic functions.

Then the theorem to be proved is

```
Q, iswfexpr e ≡ TT, isname n ≡ TT
⊃
∀p. iswfprog p ∧ ¬(ischanged e p) ∧ ¬(ischanged n p) ::
      (n ← e)⊗MPROG(p) ≡ MPROG((e/n)p)⊗(n ← e)
```

We have proved this theorem on the computer. It was lengthy - about 850 proof-steps not counting the axioms Q, which took a further 50 odd steps - but the experience has shown us how the proof may be shortened drastically by introducing a conceptually simple but powerful simplification mechanism to be invoked at will by the user. Also, less than half of the steps were concerned with the main theorem; the remainder were concerned with general theorems about ∨,∧,¬,= and the syntax and semantics of the programming language.

Discussion

We have emphasized formal proofs concerning the syntax and semantics of programming languages because this gives a strong motivation for the implementation of Scott's logic. Manna [8] has shown that certain properties of programs are representable as the satisfiability or validity of certain formulae in the first-order Predicate Calculus, but the task of deriving the appropriate formula in each case is not carried out in the Predicate Calculus. Also Burstall [10] explored the possibility of representing the syntax and semantics of ALGOL programs within first order logic. However, since the Predicate Calculus does not handle partial functions with ease, it cannot naturally express the function computed by an arbitrary program. This remark does not disparage the theoretical value of Manna's and Burstall's results; it suggests that for the complete formalization of program proving, the Scott logic is a more natural tool.

We have given LCF a full type-structure. However, Scott's work on continuous lattices [13] shows that we should proceed to a version of the logic that is type-free.

Another important next step is to explore further how to represent the formal semantics of programs within the logic; our example was of a very simple language, and we would like to show how the more realistic formal semantics of Strachey (see [11] and his recent work as yet unpublished) and of the PL/1 group [12] can be handled in the logic. Last but not least, we propose to formalize proofs of compiler correctness, again drawing on those techniques already in existence.

Acknowledgements

My debt to Dana Scott's work is very clear. Also I am grateful to John McCarthy and Richard Weyhrauch for many helpful discussions and ideas.

References

1. Scott, D., "A Type-theoretical Alternative to CUCH, ISWIM, OWHY", (Unpublished) Oxford (1969).
2. Milner, R., "LCF - Logic for Computable Functions: an implementation", forthcoming A.I. Memo, Computer Science Department, Stanford (1971).
3. Enea, H., "MLISP", Report CS-92, Computer Science Department, Stanford (1968).
4. Smith, D.C., "MLISP", Memo AIM-135, Computer Science Department, Stanford (1970).
5. McCarthy, J., "A Basis for a Mathematical Theory of Computation", Computer Programming and Formal Systems, pp 33-70 (eds. Braffort P., and Hirschberg, D.,) North Holland (1963).
6. de Bakker, J.W., and Scott, D., "A Theory of Programs", (Unpublished) Vienna (1969).
7. McCarthy, J., "Predicate Calculus with "undefined" as a truth-value", Memo AIM-1, Computer Science Department, Stanford (1963).
8. Manna, Z., "Properties of Programs and the First-Order Predicate Calculus", JACM Vol. 16, No. 2, pp 244-255 (1969).
9. McCarthy, J., "Towards a Mathematical Science of Computation", Information Processing; Proceedings of IFIP Congress 62, pp 21-28, (ed. Popplewell, C.M.), Amsterdam, North Holland (1963).
10. Burstall, R.M., "Formal Description of Program Structure and Semantics in First-Order Logic", Machine Intelligence 5, (eds. Meltzer, B., and Michie, D.) Edinburgh University Press (1969).
11. Strachey, C., "Towards a Formal Semantics", Formal description Languages for Computer Programming (ed. Steel, T.B., Jr.), Amsterdam, North Holland (1965).
12. Lucas, P. and Walk, K., "On the Formal Description of PL/1", Annual Reviews in Automatic Programming 6,3 (1969).
13. Scott, D., "Outline of a Mathematical Theory of Computation", Proc. 4th Princeton Conference on Information Science and Systems (1970).