A CONSTRUCTIVE THEORY OF RECURSIVE FUNCTIONS[†]

Robert L. Constable

TR 73-186

October 1973

Department of Computer Science
Cornell University
Ithaca, New York 14850

[†]This work was supported in part by the National Science Foundation Grant GJ-5979.

CONTENTS

§1 A language with computational meaning

(1.1) The computer scientist confronting a theorem of classical mathematics must sort out several important questions. If the theorem says, "For all x there is a number y such that ..." he may need to know whether the number y "really exists", i.e. can be computed or whether it is known only that "it is impossib that such a number not exist". In one case, the theorem will give him a number, in the other he is left with a contradiction. It is difficult to compute with contradictions.

If you think the situation is contrived, ask your local numerical analyst how he really uses what is reputed to be "one of the most useful tools of the calculus", the Mean Value Theorem.†

So the computer scientist asks his colleagues the mathematicians to save computing people a lot of trouble by being careful to distinguish between "there exists" and "we can find", say use the symbols $\forall_x$ and $\exists x$, respectively.

---

†The question is, does he use the

(a) classical mean value
assume f is differentiable on [a,b]; then <u>there is a point</u> $x_0 \in [a,b]$ such that
$$f(b)-f(a) = f'(x_0)(b-a)$$
or

(b) constructive mean value
assume f is differentiable on [a,b]; then for every ϵ <u>we can find</u> $x \in [a,b]$ such that
$$|f(b)-f(a)| - f'(x)(b-a)| < \epsilon .$$

But now the algorist[†] confronts arguments of the type

> For all x, if P(x) then we can find y_1
> if $\neg P(x)$ then we can find y_2
> Since $P(x) \vee \neg P(x)$, we can always find some y
> (either y_1 or y_2).

This kind of theorem is unfortunate because given x an algorist may not know whether P(x) is true or $\neg$ P(x) is true, so he really cannot find a y . Therefore he asks his friends to be careful whether they mean by P(x) either

(a) P(x) is true or

(b) P(x) is known (proved)

because then the argument might read

> For all x, we can either prove P(x) or prove $\neg$ P(x), and if P(x) then we can find y_1, else we can find y_2. Therefore for all x we can find some y.

Of course one difficulty in meeting this request is that there is a tremendous amount of work involved in sorting out this language for all existing mathematics. Is the work justified to merely please those people interested in computation?[††]

[†] "Algorist" is the old name for "computing person."

[††] Another difficulty for the classical mathematician is that if he carefully organizes his work to separate the language of "P is known" and "we can find" from the language of "P is true" and "there exists", then people may lose interest in the theorems about truth and existence. At best they may ask embarrasing questions about "truth."

The work certainly would not be justified if there were few interesting and deep theorems which could be stated using the newly distinguished primitives, "we know" and "we can find." In fact, there is a widespread belief among mathematicians that those thoerems which can be formulated in the constructive language are not as interesting as the other theorems (perhaps,they feel, because "we know" so little and so much "is true").†

This widespread opinion is wrong as Bishop [1] has shown. Nevertheless the argument about what "is interesting" is unfair. The computer scientist is not asking mathematicians to throw away anything. He merely wants a more careful attention to computational meaning so that he can choose what is interesting to him.

To begin this paper, let us systematically examine a language based on "we know" and "we can find," and then use it to review foundations of computing theory.

(1.2) logical connectives

Let A and B be sentences. Then we consider symbols for connecting sentences.

---

†This belief was questioned less than it should be because it was so vehemently held by David Hilbert in response to the intuitionist Brouwer's attacks on classical mathematics. Brouwer contended that such mathematics was just plain wrong and should be thrown away (a view certain to provoke over-reaction).

Definition 1: (i) Negation, $\neg$

$\neg$ A means that A is not a fact, that is, if we assume that A can be proved, then we can prove a contradiction in mathematics, specifically that 0 = 1. We can also take the stronger definition that $\neg$ A means A can never be proved.

Sometimes we let $\sim$A mean that A is a falsehood, that is $\sim$A is a truth, a true statement regardless of our state of knowledge.

(ii) Conjunction, &

A&B means that A and B are both facts, i.e. both are known by demonstration. A$\wedge$B means that A and B are truths, i.e. both are true whether we know it or not.

(iii) Disjunction, $\vee$

AorB means there is a finite procedure which will either give a proof of A or a proof of B.

A$\vee$B means that A is a truth or B is a truth. Notice that A$\vee$B can be defined as $\sim(\sim A \wedge \sim B)$, but we do not necessarily have A or B is $\neg(\neg A \& \neg B)$.

(iv) (material) Implication, =>, $\supset$

A => B means that if A is shown to be a fact, then we can show that B is a fact.

A $\supset$ B means $\sim$A$\vee$B, that is, whenever A is true, then B is true.

(v) Equivalence (biconditional), <=>, $\equiv$

A <=> B means (A => B) & (B => A). A $\equiv$ B means (A $\supset$ B) $\wedge$ (B $\supset$ A).

The connectives have a natural ordering, as do arithmetic operations, which allows unambiguous interpretation of parenthesis free expressions. Thus just as we all know that $a\cdot b+x\cdot y+c$ means $(a\cdot b)+(x\cdot y)+c$ we also know that $\neg A\&B\vee C\vee A\&C \Rightarrow \neg D$ means

$$((\neg A\&B)\vee C\vee(A\&C))\Rightarrow \neg D.$$

The precedence rule is that $\Leftrightarrow$ gets its arguments first, then $\Rightarrow$, then $\vee$, then $\&$ and finally $\neg$. For $\&$, $\vee$, $\neg$, $\Leftrightarrow$ the left to right precedence is immaterial, but for $\Rightarrow$ the rule is that $A \Rightarrow B \Rightarrow C$ means $(A \Rightarrow B) \Rightarrow C$, i.e. we associate from the left. For example $A \Rightarrow B\vee C\&\neg A\vee B\Leftrightarrow A\Leftrightarrow B\vee C$ is

$$((A\Rightarrow(B\vee((C\&\neg A)\vee B)))\Leftrightarrow A)\Leftrightarrow(B\vee C)$$

(1.3) Logic of quantifiers and predicates

If A is a sentence like "x is even" which makes a statement about an individual variable x, then we call A a predicate or a relation. These formulas need quantifiers to make them into sentences. The constructive quantifiers are $\exists$, $\forall$. The classical quantifiers are $\vee$, $\wedge$.

Definition 2: (i) Existential quantifiers

$\exists x A(x)$ means, "we can find an object x_0 such that $A(x_0)$ is provable."

We sometimes use $\vee_x A(x)$, "there is an object x_0 such that $A(x_0)$ is a truth."

The symbol $\vee$ is used because in a finite universe, say $D = \{d_1,\ldots,d_n\}$, to say "there is an x such that A(x)" means precisely $A(d_1)\vee A(d_2)\vee\ldots\vee A(d_n)$ which can be written $\bigvee_{d\in D} A(d)$.

(ii) Universal quantifiers

$\forall x A(x)$ means that there is a procedure (depending only on x) for taking any object x and verifying that A(x) is provable.

$\bigwedge_x A(x)$ means $\sim\bigvee_x \sim A(x)$ or, "for all x, A(x) is true".

The notation, $\bigwedge_x A(x)$ is explained by the fact that for a finite domain D, $\bigwedge_{d \in D} A(d)$ means $A(d_1) \wedge A(d_2) \wedge \ldots \wedge A(d_n)$.

(1.4) Formal rules

There are well-known algorithms for deciding whether a logical expression E involving classical connectives is true under all classical interpretations (i.e. whether E is a tautology).

One algorithm, the classical truth table method is trivially seen to be correct, but it requires exponentially many steps to execute (as a function of the number of variables). The method consists merely of filling in the truth table for a formula and checking whether all possible assignments yield "truth". An example should suffice to explain the method.

P	q	$p \supset q$	$\sim q \supset \sim p$	$p \supset q \supset (\sim q \supset \sim p)$
T	T	T	T	T
T	F	F	F	T
F	T	T	T	T
F	F	T	T	T

To apply the method one needs only to know the following truth table definitions of the classical connectives.

p	q	∿q	p∧q	p∨q	p ⊃ q	p ≡ q
T	T	F	T	T	T	T
T	F	T	F	T	F	F
F	T	F	F	T	T	F
F	F	T	F	F	T	T

Another algorithm for classical tautologies is the tableaux (or tree) method. The idea of this algorithm is to search for a falsifying truth assignment in a systematic manner. Given a formula such as

$$\sim(p \vee q) \supset p \wedge \sim q$$

we attempt to make it false. That is we look for a way to satisfy the signed formula

*1 $F \sim(p \vee q) \supset (p \wedge \sim q)$.

An implication $A \supset B$, can be false only if $A = T$ and $B = F$. So we say that *1 derives the line *2

*2 $T\sim(p \vee q)$, $F(p \wedge \sim q)$.

Now $\sim(p \vee q)$ can be <u>true</u> in only one way, $p \vee q$ is false. Thus *2 gives

*3 $F(p \vee q)$, $F(p \wedge \sim q)$.

But $(p \vee q)$ can be false only if both p and q are false, thus

*4 Fp, Fq, $F(p \wedge \sim q)$.

But F(p ∧ ∿q) (read "false p and ∿q") can occur in two ways. Fp or F ∿ q. Thus we must consider a branch in the enumeration of cases. We obtain

*5 Fp, Fq, Fp ∧ ∿ q

Finally F ∿ q iff Tq. So we obtain the two branches

*6 Fp,Fq,Fp | Fp, Fq, Tq

The right branch is clearly contradictory because we have Fq and Tq. Thus it can not happen. But the left branch says that if we assign F to p, F to q, then we will obtain an F for the original formula, thus in the truth table there will be a line

P	q	p ∨ q	∿(p ∨ q)	p ∧ ∿ q	∿(p ∨ q) ⊃ (p ∧ ∿q)
F	F	F	T	F	F

an the original formula is not a tautology.

In order to form a binary tree of the type given in the example we use the rules

(1) $\dfrac{T\sim X}{FX}$ $\qquad$ $\dfrac{F\sim X}{TX}$

(2) $\dfrac{T(X\wedge Y)}{\begin{array}{l}TX\\TY\end{array}}$ $\qquad$ $\dfrac{F(X\wedge Y)}{FX \mid FY}$

(3) $\dfrac{T(X \vee Y)}{TX \mid TY}$ $\qquad$ $\dfrac{F(X \vee Y)}{FX \\ FY}$

(4) $\dfrac{T(X \supset Y)}{FX \mid TY}$ $\qquad$ $\dfrac{F(X \supset Y)}{TX \\ FY}$

A rule of the type $\dfrac{T(p \supset q)}{Tq \mid Fp}$ means, if $T(p \supset q)$ appears at a node along with a set of other Formulas S, then form a branch in the tree and put S, Tq on one branch and S, Fp on the other.

When the tree has reached the condition that all formulas on all nodes are of the form Fp_i or Tp_i for basic letters p_i, then the expansion stops. If a branch contains a contradiction, Tp, Fp, then it is called closed. If every branch is closed, then the tree is closed and the formula F(X) which formed the root can not be made false. Otherwise, every open branch gives a truth assignment of false to formula X.

There is a tableaux algorithm to decide the constructive validity of formulas. The algorithm is similar to that above except that (a) at certain nodes, only a subset of the signed formulas is carried to a new node and (b) there may be several different ways to expand a node so that a forest of trees is grown by the algorithm instead of a single tree. The original signed formula, FX, can not be satisfied as long as there is one closed tree in the forest. (Each tree represents a possible evolution of knowledge and a formula X is true only if it is true in any possible evolution of knowledge.) Thus a closed

tree in the forest is a proof of X.

To state the rules, let S be a set of signed formulas (e.g. {T(p ∨ q), F(p & q), Fp}). Then let S_T be the subset of formulas with sign T, i.e. the known truths (e.g. {T(p ∨ q)}). The tree growing rules, taken from Fitting [5], are

(1) $\dfrac{S,T(\neg X)}{S,FX}$ $\qquad$ $\dfrac{S,F(\neg X)}{S_T,TX}$

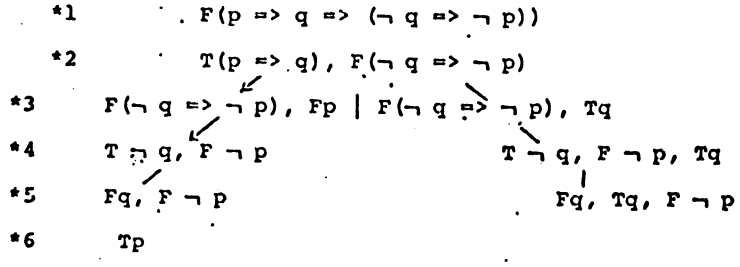
(2) $\dfrac{S,T(X\&Y)}{S,TX,TY}$ $\qquad$ $\dfrac{S,F(X\&Y)}{S,FX\,|\,S,FY}$

(3) $\dfrac{S,T(X\text{or}Y)}{S,TX\,|\,S,TY}$ $\qquad$ $\dfrac{S,F(X\text{or}Y)}{S,FX,FY}$

(4) $\dfrac{S,T(X{=>}Y)}{S,FX\,|\,S,TY}$ $\qquad$ $\dfrac{S,F(X{=>}Y)}{S_T,TX,FY}$

The rules are used as before, but now it is possible to alter the trees significantly by changing the order of applying the rules. Consider the following example carefully.

We want to know whether (p => q) => (¬ q => ¬ p) is valid constructively, so we try to find a counterexample,



It appears that the formula does have a falsifying instance because this tree is not closed. But this is misleading, it is ou selection of the tree which is at fault.

Notice the following different derivation.

**1 F(p => q => (¬ q => ¬ p))

**2 T(p => q), F(¬ q => ¬ p)

**3 T(p => q), T ¬ q, F ¬ p

**4 T(p => q), T ¬ q, Tp

**5 Fp, T ¬ q, Tp | Tq, T ¬ q, Tp

**6 Tq, Fq, Tp

This is a closed tree for the formula. This means that F(p => q => (¬ q => ¬ p)) is not always satisfiable. We regard the closed tree given above as a proof of

p => q => (¬ q => ¬ p).

Here is another example of a formal constructive proof of validity in the constructive propositional calculus.†

F(¬ ¬ (p or ¬ p))

T(¬ (p or ¬ p))

T(¬(p or ¬ p)), F(p or ¬ p)

T(¬(p or ¬ p)), Fp, F ¬ p

T(¬(p or ¬ p)), Tp

---

†To be precise about this deduction we should have the rule $\frac{S,x}{S,x,x}$. This allows multiple copies of a statement.

F(p or ¬ p), Tp

Fp, F ¬ p, Tp

X

Notice that in step 3 we keep the set of true formulae around even after deriving its consequence.

The tableaux algorithm is extended to the constructive predicate calculus by adding the following rules:

T∃	$\frac{S,\ T\ \exists x\ S(x)}{S,\ TS(a)}$	a is a new variable
F∃	$\frac{S,\ F\ \exists x\ S(x)}{S,\ FS(a)}$	
T∀	$\frac{S,\ T\ \forall x\ S(x)}{S,\ TS(a)}$	
F∀	$\frac{S,\ F\ \forall x\ S(x)}{S_T,\ FS(a)}$	a is new

When we say that "a is new" we mean it has not been used in any of the formulas of S.

(1.5) Sample proofs

To convey the spirit of constructive proofs, we present a few simple examples from number theory. Our axioms for number theory are Peano's axioms. We also assume

Constructivist Thesis: Every integer can be transformed by a rule into decimal form.

Definition: Say a divides b , a|b , iff we can find a number c such that a·c = b. Write a∤b for ¬(a|b).

Division Algorithm Theorem:

Given integers a,b, with $b > 0$, we can find unique integers q and r such that $a = bq + r$, $0 \leq r < b$. We write $a \div b = q$ and $rm(a,b) = r$.

Proof:

If $b > a$, then $q = 0$ and $r = a$. If $b = a$, then $q = 1$ and $r = 0$. To tell whether $b > a$ or $b = a$, transform each number to decimal form and compare in the usual way. If $b < a$, then let q be the least x such that $b \cdot x \geq a$. (We guarantee this q and a procedure to find it from a form of the induction axiom. Namely if we know P(a) then we can find a least n such that P(n).)

Put $r = a - b \cdot q$. Clearly $r < b$; otherwise q is not the least x. Q.E.D.

Definition: Say p is prime iff $p > 1$ and for all c, $1 < c < p$, $\neg (c \mid p)$. Write Prime(x) iff x is a prime.

Theorem: (a) $\bigwedge_x (\text{Prime}(x) \vee \sim \text{Prime}(x))$

(b) $\forall x$ (Prime(x) or $\neg$ Prime(x))

Proof:

The classical statement (a) is a theorem of logic. To prove it one uses the axiom that $S(x) \vee \sim S(x)$ for any predicat

The constructive statement (b) requires a proof based on the facts of arithmetic. In fact the proof gives a decision procedure for the predicate Prime ( ).

We prove

$$\exists y(1 < y < x \;\&\; y \mid x) \text{ or } \neg \exists y(1 < y < x \;\&\; y \mid x)$$

To prove a disjunction A or B we give a procedure to prove A or to prove B. In this case the procedure is to try $y \mid x$ for $y = 2,3,\ldots,x-1$. The procedure is clearly finite since to decide $y \mid x$ one computes $rm(x,y)$ by the Division Algorithm. The procedure either produces a y or else allows us to conclude $2 \nmid x$, $3 \nmid x$, $\ldots$, $(x-1) \nmid x$ from which we prove $\neg \exists y(1 < y < x \;\&\; y \mid x)$. Q.E.D.

As a final illustration we prove Euclid's Theorem on primes.

Theorem : There are infinitely many primes, i.e.

$$\forall x \exists p \; (x < p \;\&\; \text{Prime}(p)).$$

Proof:

We show that for any x, there is a prime number $p > x$. We claim there is a prime number p such that $x < p \leq x! + 1$ where $x! := x \cdot (x-1) \cdot (x-2) \cdot \ldots \cdot 2 \cdot 1$.

Let y_1 be the smallest $y \leq x! + 1$ which divides $x!+1$ (y_1 may be $x!+1$). We claim that y_1 is a prime and $x < y_1$. If y_1 is not a prime, then there is a $y_2 < y_1$ such that $y_2 \cdot c = y_1$ and since $\exists d \; y_1 \cdot d = x!+1$ it is clear that $y_2 \mid (x!+1)$. But this contradicts our choice of y_1. Hence y_1 is prime.

Notice that for all $1 < y < x$, $y \nmid (x!+1)$ because $y \mid x!$ Hence $x < y_1$. Q.E.D.

Other examples of constructive proof can be found in Kleene [7] and Heyting [6].

(1.6) Set theory

Introduction

Increasingly since the late 1800's classical mathematics has been expressed in the language of set theory. So far the language has proven adequate for the expression of all known mathematics. Today many students of mathematics may think that mathematics <u>is</u> the study of sets, and undoubtedly the concept of set is fundamental as developments of the last century have shown.

However, in constructive and computational mathematics the concept of set is not as fundamental as the concepts of number and rule. This is because the general notion of set does not have the computational content of numbers and functions. For example, the set R of real numbers does not enter into computations. It has no concrete representation and it is a little "fuzzy around the edges" because we cannot decide when a sequence of rationals is in fact a real number.

There are on the other hand sets which can be used in calculations, and set theoretic concepts are efficient ways to describe computations. So in fact certain parts of set theory are indispensible for computing work.

Moreover, if we are to use the great edifice of 20th century mathematics, then we must be able to use the language of set theory constructively. So we shall mention the basics of this theory here.

Constructive set theory is not yet highly developed. It differs greatly from the classical theory, especially on the

matter of complements, cardinal and ordinal numbers. Nevertheless computer science is able to survive rather well with it even on these topics because it does not need the highly developed parts of the theory.

In a naive way we think of a set (or a species) as a collection of mathematical objects. When we attempt to be more precise about sets, we follow the axiomatic approach. That is, we do not say what a set is, rather we say how to talk about membership in sets, subsets, unions, products, etc. and of course how to specify or define a set.

Definition 1: To define a set S we must say how to construct an element of S and how to tell when two elements of S are equal.

One of our basic names for finite or subfinite sets is simply a list of elements, $\{e_1, e_2, \ldots, e_n\}$. Notice that 1 is distinct from $\{1\}$.[†] Furthermore, for discrete sets, when no equality relation is explicitly mentioned, e_i and e_j for $i \neq j$ are assumed to be distinct. Thus $\{1,1,2,2\}$ is not the usual way to represent the subset of integers $\{1,2\}$ with the usual equality relation. This means that the set $\{e_1, e_2, \ldots, e_n\}$ without explicit mention of equality is a finite set of n elements.

[†] One might think it advisable to eliminate unit sets and demand that all sets have at least two objects. This would eliminate this peculiar distinction between $\{1\}$ and and 1. However, if we eliminated unit sets, then we could not use the term set freely. If we did not know whether a certain equation had more than one root, for example, we could not speak of the set of all roots of the equation.

Definition 2: To prove that x is a member of a set S, written $x \in S$, we must verify that x is one of the constructions specified in the definition of S.

Definition 3: Given a set S, a subset S' is defined by giving conditions for a construction in S to be in S'. The conditions are given by a predicate or relation on S. The standard notation for a subset is $S' \subseteq S$ and a standard description is

$$\{x \mid P(x) \ \& \ x \in S\}$$

which reads, "the set of all x such that P(x) and x belongs to S." For example

$$\{2 \cdot x \mid x \in \mathcal{N}\} = \{x \mid x \in \mathcal{N} \ \& \ x \text{ is even}\}$$

defines the subset of even numbers.

Examples:

Besides the discrete finite sets $\{e_1, \ldots, e_n\}$ we have seen the following examples of sets:

(1) $\mathcal{F}_{\mathcal{N}}$ number theoretic functions: to construct an element give a rule from integers to integers and prove that for all $x \in \mathcal{N}$ the result of applying the rule is a unique $y \in \mathcal{N}$; two functions f_1, f_2 are equal iff $f_1(x) = f_2(x)$ for all $x \in \mathcal{N}$. Notice, equality on $\mathcal{F}_{\mathcal{N}}$ is not the same as identity on the set of algorithms. This in essence makes a function into a set of ordered pairs.

(2) $\mathcal{P}(\mathcal{N})$ power set of $\mathcal{N}$: to construct an element of $\mathcal{P}(\mathcal{N})$, define a subset of $\mathcal{N}$; two subsets A, B are equal iff $A \subseteq B \;\&\; B \subseteq A$.

(3) $\mathbb{R}$ constructive numbers, to constrcut an element, construct a sequence of rationals such that $|x_n - x_m| < 1/n + 1/m$; two elements $x = \{x_i\}$ and $y = \{y_i\}$ are equal iff $|x_i - y_i| < 2/i$.

The principle that any well-defined relation P( ) on a set S determines a subset, $S_p = \{x | x \in S \;\&\; P(x)\}$ is called the comprehension principle.

We write $S' \subset S$ if $S' \subseteq S$ and $S \neq S'$ and say S' is a proper subset.

Definition 4: The set of all subsets of x is called the power set of x and is denoted $\mathcal{P}(x)$. The set of all functions from x to (0,1) is denoted 2^x, and is called the free power set.

Remark: For finite sets there is an obvious correspondence between $\mathcal{P}(x)$ and 2^x, namely the characteristic function of a subset y belongs to 2^x and every function in 2^x is such a characteristic function.

This correspondence does not hold for arbitrary sets. In particular $\mathcal{P}(\mathcal{N})$ and $2^{\mathcal{N}}$ are not the same (i.e. are not known to be the same).

Remark: In a naive classical set theory one can imagine some paradoxical sets. For example, let $u := \{x | x \text{ is a set and } x \notin x\}$, e.g. u is the set of all sets which do not contain themselves as elements (one would imagine u to be very large, perhaps the entire universe, hence the letter u). But now one asks whether $u \in u$. If $u \in u$, then by definition $u \notin u$. On the other hand, $u \notin u$ implies by definition of u that $u \in u$. We thus have

$$u \in u \text{ iff } u \notin u .$$

According to classical logic either $u \in u$ or $u \notin u$, therefore classically we have a contradiction known as Russell's paradox. What is the naive situation constructively?

Constructively sets are built up in stages and it is not possible that any set could be a member of itself. In fact the classical mathematician's response to the Russell paradox is to consider a more constructive universe of sets and to axiomatize that universe. Here again the key idea is that sets are built up from atoms in stages. The stage at which a set is constructed is its rank.

Constructively, all sets are built up from the integers in stages. At stage 0, only the integers are given.† At stage 1 we form sets of integers. These sets are of rank 1. At stage 2

†In a pure set theory we could imagine constructing the integers from the null set ϕ. This approach has the advantage that every mathematical object is either a set or a rule, but this is no significant advantage constructively, so we take the numbers as basic elements.

we form sets whose elements can be sets of integers; these are rank 2 sets. At stage 3 we form sets whose elements can be sets of sets of integers, etc.

Definition 5: The empty set (null set, void set) denoted ϕ is the set with no elements. Thus to construct an element of ϕ, construct anything and prove $0 = 1$.

Remark: The empty set is an oddity which we attempt to avoid as much as possible. In classical set theory it can be used to build up every set.

Algebraic laws for sets

We will now turn to the simple facts we need for manipulating sets.

Definition 6: Two sets, A, B are equal iff they have the same elements regardless of how they are defined. Thus

$$A = B \text{ iff } (x \in A \text{ iff } x \in B) .$$

We say that sets are equal if they are equal in extension. For example the following sets are all equal {all even primes}, {the least positive even integer}, {all positive solutions to $x^2 = 4$}. We think of the set {2} as the extensional representation of the concepts "even prime" and "positive solution to $x^2 = 4$," and we say that the concepts are equal in extension.

The concept of extensionality is more important for us when applied to algorithms and functions. Consider the rules applied to Q, the rational numbers.

rule 1: add x to y and divide the result by 2

rule 2: divide x by 2, divide y by 2, add the results.

Both rules define the function $(x+y)/2$ from pairs of rationals.

The graph of the rule is considered to be the set of all triples $\langle x,y,z\rangle$ such that $(x+y)/2 = z$. The graph is the extension of the rule. Two rules with the same graph are said to be equal in extension. The only criterion for equality of sets is that they be equal in extension. In axiomatic set theory this important fact is called the axiom of extensionality.

Definition 7: Let x,y be subsets of a set S, then

(i) $x \cup y = \{z \mid z \in x \text{ or } x \in y\}$ union

(ii) $x \cap y = \{z \mid z \in x \ \& \ z \in y\}$ intersection

The concept of set complement is quite different constructively than classically.

Definition 8: $x - y := \{z \mid z \in x \ \& \ \neg\, z \in y\}$. We call $x - y$ the complement of y in x.

To assert $\neg\, z \in y$ one must prove that it is impossible for z to be in y. Notice that we do not have the fact $(x-y) \cup y = x$.

The classical De Morgan's laws

(i) $\overline{A \cap B} = \overline{A} \cup \overline{B}$

(ii) $\overline{A \cup B} = \overline{A} \cap \overline{B}$

and the other laws about complements such as $\bar{\bar{A}} = A$, $A \cup \bar{A} = S$ are not valid unless we assume the law of excluded middle, which of course we do not in general. Of course, $A \cap \bar{A} = \phi$ is true for all A.

Proposition 1: If for all subsets A of S, we define $\bar{A} := S - A$ and if $(S-A) \cup A = S$, then

(i) $(\overline{A \cap B}) = \bar{A} \cup \bar{B}$

(ii) $(\overline{A \cup B}) = \bar{A} \cap \bar{B}$

(iii) $A \cup \bar{A} = S$

For certain subsets of a set S we can satisfy the hypothesis of Proposition 1, and of course for finite sets S, $A \cup (S-A) = S$ for any $A \subseteq S$.

Definition 9: A subset A of S is called free (or computable), iff there is an algorithm, called the characteristic function of A, denoted $\mathscr{X}_A(\ )$ such that

$$\mathscr{X}_A(x) := \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \in S-A \end{cases}$$

Thus A is free iff there is an algorithm to decide when x is in A.

Definition 10: A set S is called discrete iff there is an algorithm to decide the equality relation among elements.

Example 1:

(1) $\mathcal{N}$ is discrete, $\mathcal{R}$ and $\mathcal{F}$ are not.

(2) All subsets of $\mathcal{N}$ of the form {1,...,n} are free.

§2 Constructive Basic Recursive Function Theory

(2.1) Introduction - formal computability

The concept of a computable function is somewhat "open-ended," as is the concept of set in informalized classical mathematics. That is, we have not ruled out the creation of wholly new formalisms for the expression of rules to define computable functions.

Let us examine the informal concept of computability specialized to number theoretic functions $f: \mathcal{N}^n \to \mathcal{N}$. Informally a function f( ) is a rule which assigns to each integer x a unique integer f(x).

We can say what a rule is by proving

$$\forall x \, \exists! y [f(x) = y]$$

because the proof will involve a procedure to find y. Then, to be more precise about rules, we become more precise about proofs.

In our informal mathematics, we can also say what a rule is by defining it. We might say:

Definition 1: A (discrete) (computing) procedure is an ordered sequence of instructions which a computing agent can mechanically and deterministically follow in a completely prescribed manner (using a prescribed data space, data and instructions) to perform some symbolic computing task. If the procedure will in principle accomplish the task in some finite amount of time measured in discrete intervals during which only one instruction is executed, then the process is a rule.

We assume that every rule can be written down, say in English. Thus if $\Sigma := \{a_1, a_2, \ldots, a_n\}$ is an alphabet containing the English letters, and the usual punctuation, then we assume any rule can be expressed in Σ^+. Thus we may speak of a rule $\alpha \in \Sigma^+$. We might refer to this assumption on the expressibility of rules as Constructivist Thesis II.

In the early 1930's logicians, motivated in part by a need to formalize the concept of a computable function, discovered a remarkable class of functions, the partial recursive functions. The first definition of this class was given by J. Herbrand in a letter to Gödel in 1931. Herbrand was a constructive mathematician, and his definition was not classically acceptable, so Gödel modified it to what is now classically known as the Herbrand-Gödel definition of recursiveness.

The class of partial recursive functions $\mathscr{PR}$ is remarkable because it appears to be an adequate formal model for the class of all computable partial functions. This appearance

emerged after it was discovered that several seemingly unrelated attempts to formalize computability arrived at the same class of functions (namely the λ-definable functions of Church and Kleene, the Turing computable functions and Kleene's μ-recursive functions). Upon philosophical and methodological analysis of the equivalence of these definitions Church and also Turing proposed the thesis that this class of partial recursive functions does indeed correspond to the informal (yet completely mathematical) notion of a computable partial function.

In this section we develop constructively the main elements of what has become known as Basic Recursive Function Theory. This is the theory of recursive functions developed from 1930 into the 1940's, mainly by Kleene, Turing and Post. We take Herbrand's definition of the partial recursive functions and then consider unsolvability and recursive enumerability.

As far as I know the constructive treatment of this theory is new, but it does not differ much from the classical theory. Some theorems, such as the assertion that $\bar{K}$ is not r.e. have a very different meaning and proof than the classical theorem. Some classical theorems, such as S is recursive iff S and $\bar{S}$ are r.e. have a surprising (and more informative) constructive formulation.

One significant new feature of the constructive theory is that it can be compared point by point to the corresponding informal theory of computable functions.

At the end of the detailed presentation of the theory, in §3, we pause to reflect on its philosophical significance,

especially on its relationship to the foundations of constructive mathematics.

(2.2) General Recursion over D

Definition 1: We consider terms formed from

individual constants	x_i	
individual variables	x_i	
function constants	f_i	and predicate constants p_j
function variables	f_i	and predicate variables p_j

A term is a constant, x_i, a variable x_i or $f_j(t_1,\ldots,t_n)$ or $f_j(t_1,\ldots,t_m)$ where t_i are terms.

A conditional term has the form

$$(\text{if } p(t_1,\ldots,t_n) \text{ then } t_T \text{ else } t_F)$$

where t_i are terms and where t_F and t_T are terms or conditional terms and where p is a predicate variable or constant.

Examples:

(1) The following are terms: x, $\underline{c}$, $f_1(x,\underline{c})$, $f_1(x,f_1(x,\underline{c}))$, $f_1(\underline{c},\underline{c})$.

(2) The following are not terms: if p(x) then x else y, f((if p(x) then x else y), f(x,c)).

(3) The following are conditional terms (if p(x) then x else y), if $p_1(x)$ then if $p_2(x)$ then $f_1(x)$ else $f_2(x)$ else $f_1(x,y)$.

A defining equation (also called equation or definition) is

$$t_1 := t_2 \quad \text{and}$$

A simple defining equation is

$$f(x_1,\ldots,x_n) := t$$

where t_1, t_2, and t are terms.

Definition 2: Given a finite sequence of simple defining equations $e_1,\ldots,e_n$ we say that the function letter on the left side of e_1 is the principal function letter. All function variables that appear only on the right hand side are called given functions. The other functions are called defined functions or auxillary functions.†

Notation: For convenience in manipulation, we shall use capital letters F_i to denote the defined function variables.

Example:

$$F_0(x) := \text{if } p_1(x) \text{ then if } p_2(x) \text{ then } F_\infty(x) \text{ else } F_1(f_1(x)) \text{ else } x$$

$$F_1(x) := \text{if } p_3(x) \text{ then } F_0(f_2(x)) \text{ else } f_1(x)$$

$$F_\infty(x) := F_\infty(x)$$

Definition 3: A system of equations E is said to define the partial function $\phi(\ )$ from $\psi_1,\ldots,\psi_n$ iff the equations E determine

---

†In some literature on the topic the given functions are called function constants and the auxillary function letters are called function variables.

ϕ uniquely when the principle function letter f represents ϕ and given function letters $g_1,\ldots,g_n$ represent $\psi_1,\ldots,\psi_n$. That is $\phi(x_1,\ldots,x_n) = y$ iff $f(\underline{x}_1,\ldots,\underline{x}_n) = \underline{y}$ can be deduced from the equations E and equations $g_i(\underline{x}_1,\ldots,\underline{x}_{n_i}) = \underline{z}$ whenever $\psi_i(x_1,\ldots,x_{n_i}) = z$.

Remark: In the classical treatment of these recursion equations, only a certain type of deduction is allowed from the equation E. Classically one would allow only three special rules

R1: Given an equation e containing the variable y, one may deduce the equation containing a constant c substituted for y.

R2: Given an equation r = s containing no variables and an equation $k(\underline{z}_1,\ldots,\underline{z}_p) = \underline{z}$ where $\underline{z}_i$, $\underline{z}$ are constants, one can deduce the equation resulting from the substitution of $\underline{z}$ for $k(\underline{z}_1,\ldots,\underline{z}_p)$ in s.

R3: Any equation of the form $g_i(\underline{x}_1,\ldots,\underline{x}_{n_i}) = \underline{z}$ can be generated if $\psi_i(x_1,\ldots,x_{n_i}) = z$.

Definition 4: A partial function $\phi:\mathcal{N}^n \to \mathcal{N}$ is partial recursive iff it is uniquely defined by a set of recursion equations E given the successor function $\lambda x[x+1]$ and $\lambda x[x=0]$.† A function

†We call ϕ non-deterministic partial recursive iff there is a system of equations E such that $\phi(x)$ is defined iff $\phi(x)=y$ is the unique solution to E at x. In Def 4 we require that the equations determine a unique value at all inputs where they determine any value.

$f: \mathcal{N}^n \to \mathcal{N}$ is recursive iff it is partial recursive and is defined at every n-tuple $\langle x_1,\ldots,x_n\rangle \in \mathcal{N}^n$.

The class of partial recursive functions is remarkable for its "invariance". As we mentioned above, the class has numerous diverse yet equivalent definitions. For the computer scientist, one of the most interesting definitions is in terms of Algol programs or some similar class of programs.

In this paper we do not precisely define a class of Algol programs or any similar class (such as RAM or RASP programs, G_3 programs), but the reader can consult [3] for such definitions. Before we state the basic result relating programs and procedures we need

Definition 5 : Given two computing procedures α,β over D, define $\alpha(x) \simeq \beta(x)$ iff $\alpha(x)$ is defined iff $\beta(x)$ is defined, and if either is defined then they are equal in D. We abbreviate the phrase"$\alpha(x)$ is defined" by $\alpha(x)\downarrow$.

Theorem:

(a) Given any set of recursion equations ϕ_i (defining partial recursive function $\phi_i(\ )$) we can find an Algol program α_j such that $\phi_i(x) \simeq \alpha_j(x)$ for all $x \in \mathcal{N}^n$.

(b) Given any Algol program α_k we can find a recusive procedure ϕ_j such that $\alpha_k(x) \simeq \phi_j(x)$ for all $x \in \mathcal{N}^n$.

(2.3) Unsolvability

A general theory of computing seeks the knowledge and methods to determine when a problem can be solved by algorithms of a certain type. It also seeks to explain the fundamental phenomena of computing experience.

In this section we discuss the cornerstones for such knowledge by presenting techniques for proving that certain mathematical problems cannot be solved by algorithms. We also prove that halting procedures cannot be separated from non-halting procedures. This explains why procedures and partial functions are inevitable in certain areas of computing theory, and alerts us to the fact that aspects of the theory will be eternally mysterious and challenging tc the human mind.

In all of this work we emphasize the underlying technique which is diagonalization and self-reference.

We begin with a simple but timely example of a numerical procedure.

Example 1: Consider the following procedure computing a partial predicate FERMAT: $\mathcal{N} \to \{1,0\}$.

FERMAT(n): [Choose an enumeration of $\mathcal{N}^3$. Let $\langle x_1(m), x_2(m), x_3(m)\rangle$ denote the m-th triple in the enumeration. On input n increase m from 0 by 1 until $x_1(m)^n + x_2(m)^n = x_3(m)^n \;\&\; x_1(m)\cdot x_2(m)\cdot x_3(m) \neq 0$].

If Fermat's longstanding conjecture is true then this procedure will halt only at n = 1 and n = 2. If the conjecture is incorrect, then the procedure will halt for other n. The best mathematical minds have failed for hundreds of years to settle Fermat's conjecture.

This example shows that deciding whether a procedure halts can be a very difficult mathematical problem. Indeed, if we believe that there will always be open mathematical problems of the form

$$\forall y P(x,y)$$

for x,y integers, then we can always find procedures whose halting is an open problem.

This example shows that we do not know whether every procedure halts or not. Since the procedure can also be defined by recursion equations, we also have:

Rejection 1: For recursion equations α_i we do not know[†]

$$\alpha_i(x)\downarrow \text{ or } \neg\alpha_i(x)\downarrow \ .$$

[†]What would it mean to say that we knew $\forall x(\alpha_i(x)\downarrow \vee \neg\alpha_i(x)\downarrow)$? It would imply the discovery of a general procedure for settling theorems in number theory. A procedure which would settle Fermat's conjecture, Goldbach's conjecture, the prime pairs problem, the four color problem, etc. all in a uniform way. The procedure might only exist "in principle". That is, it might be so expensive to apply that we could never use it to actually settle these questions.

So this rejection is a convincing argument that any attempt to determine whether or not an arbitrary recursion equation is totally defined will be very difficult. However, we cannot be sure from these considerations alone that the task is impossible.

Suppose we look harder at this problem of determining when a procedure is defined. Could we actually show that it is absolutely impossible to solve all problems of the form

$$\alpha_i(x)\downarrow \vee \neg\, \alpha_i(x)\downarrow \ ?$$

Evidence from Example 1 suggests that solving the problem is impossible, but to prove that it is impossible we would have to argue about the class of all possible algorithms. It seems unlikely that we could "grasp this class mathematically" because it is a "creative class." That is, as human minds create new expressions they alter the known representations of rules. Thus there is no fixed concrete representation of the class of all rules, as there is for the class of integers.

However, unlikely as it may seem, we can indeed prove that it is impossible to solve this problem.

Recall that if α is a procedure, then it can be expressed in English, say in Σ^*. Its location in the standard enumeration of Σ^* is called its index and is denoted i.

Theorem 1: There does not exist a rule

$$h: \mathbb{N} \to \{0,1\}$$

$$h(i) = \begin{cases} 1 & \text{if there is a procedure } \alpha,\ i_\alpha = i \text{ and } \alpha(i_\alpha) \text{ is defined} \\ 0 & \text{otherwise .} \end{cases}$$

Proof: Since this is a negative assertion we can prove it by contradiction. Assume such a rule h (mnemonic for "halting") exists. Then define a new rule, d (mnemonic for "diagonal"):

L1: IF h(i) = 1 THEN L1 ELSE HALT(1) FI[†]

Thus if h(i) = 1 the procedure loops forever, otherwise it stops with answer 1.

Clearly d is a rule, it has an index, i_d. Also since h is a rule, it gives value 0 or 1 on $h(i_d)$. We show that this value is always incorrect.

If $h(i_d) = 1$, then by definition of d, $d(i_d)$ loops forever.

If $h(i_d) = 0$, then by definition of d, $d(i_d) = 1$. Each possibility is wrong, so h is wrong.

Q.E.D.

Remark: Notice not only is h wrong, but we can find out exactly where h must fail, namely on the integer i_d. We can find i_d because d is constructed from h.

It may seem that the difficulty here is that h must try first to decide which integers are indices of procedures.

But the same type of argument works even if we do not require that h decide this. Suppose we always agree to supply h with a proof that its input integer is the index of a procedure, and suppose we do not care what h does on the other integers. Thus we allow a partial function $\beta: \mathcal{N} \to \{0,1\}$ for h. We can prove the same result. We can also prove the result by considering procedures which operate directly on Σ^*.

[†] "FI" is a delimiter matching "IF" which we have adopted from Algol 68.

Theorem 2: There does not exist a procedure $\beta: \Sigma^* \to \{0,1\}$ which will determine whether an arbitrary computing procedure halts on its own index.

Proof: To prove a negative assertion we can proceed by contradiction. Suppose such a procedure exists, call it β. Then by definition

$$\beta(\alpha) := \begin{cases} 1 & \text{if } \alpha(\alpha)\downarrow \text{ for } \alpha \text{ a procedure} \\ 0 & \text{otherwise .} \end{cases}$$

Notice β need only be defined on procedures not on all strings. So domain $\beta := \{\alpha \mid \alpha \text{ is a procedure}\}$.

Now define a new procedure with the same domain as β.

δ: [On input x compute $\beta(x)$ where $x \in \Sigma^*$
If $\beta(x) = 1$ then repeat the rule (i.e. loop)
If $\beta(x) = 0$ then output a 1]

This is a procedure once we are given β. Notice if $\beta(x)$ is undefined then so is $\delta(x)$. Also given β expressed in Σ^* we can find the expression of δ in Σ^*. Now compute $\delta(\delta)$.

Since δ is a procedure, β is defined on δ, Thus by definition $\beta(\delta)$ is either 0 or 1. We show that β is wrong in either case.

If $\beta(\delta) = 1$, then $\delta(\delta)$ loops and does not define a value. So $\beta(\delta)$ should be 0. If $\beta(\delta) = 0$, then $\delta(\delta) = 1$ and is defined. So $\beta(\delta)$ should be 1. Thus β is always incorrect on procedure δ. So no correct β exists.

Q.E.D.

We thus know that a certain mathematical problem cannot be solved. In fact we know that there are statements of the type

$$\forall x(P(x) \vee \neg P(x))$$

which cannot be proved because to prove them we need a rule to decide P and for certain P there is no rule.

We now have a theorem of informal constructive logic which contradicts a classical theorem. Namely,

Fact 1 : $\neg \forall x(P(x) \vee \neg P(x))$.

(2.4) reducibility

With one wave of the hand we can generate a plethora of unsolvable problems from this one example. The appropriate magic trick is ignoring the input.

Consider the following problems.

(1) Is numerical procedure α defined at 0?

(2) Does procedure α halt on any integer?

(3) Does procedure α halt on all integers?

(4) Are procedures α_1, α_2 equivalent on $\mathcal{N}$; e.g. $\alpha_1(x) \simeq \alpha_2(x)$ for all x?

Theorem 1 : None of the problems (1) - (4) is solvable.

Proof: We prove each problem unsolvable in the same way. We show that if it were solvable, then the question of whether α halts on its index would be solvable.

(1) Suppose there is a procedure, β_0, to decide whether $\alpha(0)\downarrow$ Thus

$$\beta_0(i_\alpha) := \begin{cases} 1 \text{ if } \alpha(0)\downarrow \\ 0 \text{ otherwise} \end{cases}$$

Then given procedure α, convert it to a procedure α' which does the following:

[On input x, compute $\alpha(i_\alpha)$].

Thus $\alpha'(x) \simeq \alpha(i_\alpha)$ for all x.

Now clearly $\alpha'(0)\downarrow$ iff $\alpha(i_\alpha)\downarrow$. Thus

$$\alpha(i_\alpha)\downarrow \text{ iff } \beta_0(i_\alpha,)\downarrow$$

Therefore β_0 would solve the problem of whether $\alpha(i_\alpha)\downarrow$, but this problem is unsolvable, so no such β exists.

Problems (2) and (3) can be shown unsolvable by the exact same argument. In case (2), for example,

$$\exists x(\alpha'(x)\downarrow \text{ iff } \alpha(i_\alpha)\downarrow).$$

To show that (4) is unsolvable, we choose a procedure θ which is undefined at all arguments, $\theta(x) = \Omega$ for all x. Then if there is a rule β such that

$$\beta(i_\alpha,i_\beta) = \begin{cases} 1 & \text{if } \alpha(x) \simeq \beta(x) \text{ for all } x \\ 0 & \text{otherwise} \end{cases}$$

then $\beta(i_\alpha,i_0) = 1$ iff $\alpha(x) = \Omega$ for all x, so that $\beta(-,i_\theta)$ solves problem (2).

Q.E.D.

Corollary 1 : The set of rules is not a computable subset of Σ^*.

Proof: This follows from the unsolvability of problem (3).

Corollary 2: The set of partial functions $\phi: \mathcal{N} \to \mathcal{N}$ is not discrete.

Proof: The equality notion for functions is, $\phi(\ ) = \psi(\ )$ iff $\phi(x) \simeq \psi(x)$ for all x. By problem (4) this is not solvable.

Remark: Notice that the set R of all names of rules is a discrete set because $\alpha = \beta$ iff α and β are the same strings in Σ^*.

The method behind the proof of Theorem 1 in §2.3 is illustrated by the following:

Theorem 2 : The set of number theoretic functions $f: \mathcal{N} \to \mathcal{N}$ is not enumerable.

Proof: Since this is a negative statement we can prove it by contradiction. To this end assume that the functions are enumerable, say as $f_1(\ ), f_2(\ ),\ldots,f_n(\ )\ldots$. Then define the function

$$f(x) := f_x(x) + 1 .$$

This is clearly a function, but it is not in the proposed enumeration because if it were, say at location d, then

$$f(d) = f_d(d) + 1 = f(d) + 1.$$

Q.E.D.

We can make an even stronger assertion.

Theorem 3: The set of rules R is not enumerable.

Proof: Assume that $a_0, a_1, \ldots, a_n, \ldots$ is an enumeration of algorithms. To know that a_i is an algorithm we would have a proof that

$$* \quad \forall i \, \forall x \, \exists y (a_i(x) = y).$$

This means that we know how to compute $a_i(x)$ for all x. Thus the following is a rule μ:

[Given i,x find the y such that $a_i(x) = y$ and output y. To find y use the proof of *.]

Now μ computes $\mu(i,x) = a_i(x)$ which is impossible by Theorem 2.

Q.E.D.

(2.5) recursive unsolvability

The proof we used in (2.3) to exhibit absolutely unsolvable problems will also suffice to show the existence of recursively unsolvable problems. The results can be phrased somewhat differently becuase we now have an enumeration of all possible recursion equations. (Note, we do not know an enumeration of all possible computing procedures.) We assume that the reader knows how to enumerate all possible sets of recursion equations, say $\hat{\phi}_0, \hat{\phi}_1, \ldots$:

Theorem 1 : There is a recursive enumeration of all recursion equations, We denote this as $\hat{\phi}_0, \hat{\phi}_1, \hat{\phi}_2, \ldots$.†

Remark: Thus to each set of recursion equations ϕ_i we can assign an index, i, just as with every procedure α we can find its index, i_α. But now conversely, given any integer n ϵ , we can recursively find a set of equations ϕ_n. This is something we are unable to do for computing rules as defined in Definition 1 of §2.1. p. 24.

Now exactly as in section (2.1) we prove:

Theorem 2 : There is no recursive function h( ) such that

$$h(i) = \begin{cases} 1 & \text{if } \phi_i(i) \text{ is defined by some deduction} \\ 0 & \text{otherwise.} \end{cases}$$

We read this as saying, the Halting Problem on Index for recursion equations is recursively unsolvable.

To phrase this in terms of sets we define K:= $\{i | \phi_i(i) \text{ halts}\}$. We then have proved:

Corollary: K is not a recursive set.

---

†This is an enumeration of all non-deterministic partial recursive functions. To enumerate the usual partial recursive functions we must define the principal deduction from a system of equations.

We are also able to duplicate the reducibility results of section (2.4) for the notion of recursive reducibility.

(2.6) enumerable and recursively enumerable sets

In this section we examine the notions of computability, recursiveness and unsolvability for sets. In some respects these topics are more subtle for sets than for functions because sets are more illusive mathematical objects. The concept of an enumerable set is especially intriguing because it lies on the border between the computable sets, which we can grasp, and non-computable sets, which we cannot.

In the realm of computable sets, the basic concepts are simpler than in the case of functions because such a set can be viewed as a (0,1)-valued function. Indeed we can view such a set as a path through the infinite binary tree.

We begin by recalling basic definitions.

Definition 1: A set S is computable iff there is an algorithm to decide membership in it.† A set S is recursive iff there is a recursive function to decide membership in it.

A set S is enumerable iff we can find a function $f: \mathcal{N} \to S$ onto S. It is recursively enumerable (r.e.) iff $f(\ )$ is recursive.

Remark: We do not know Church's Thesis to be valid, so we cannot assume that computable and recursive sets are the same.††

---

†Bishop's term for such sets is "free."

††See §3 for a discussion of this thesis and its status in constructive mathematics.

Also we have not specified that S be a set of integers. We can in fact allow as members of S any mathematical objects to which an algorithm can be applied.

We begin gathering information about enumerable and recursively enumerable sets with the following simple observations. The result is simple so that the reader can concentrate on certain important proof techniques used frequently in this section.

Theorem 1 :

(a) If S is a discrete infinite subset of $\mathcal{N}$, then S is equipotent with $\mathcal{N}$ iff S is enumerable.

(b) If S is a recursively discrete infinite subset of $\mathcal{N}$, then S is recursively enumerable iff S is recursively equipotent with $\mathcal{N}$.

Proof: To prove (a) we must define a 1-1 map from $\mathcal{N}$ onto S. The idea is elementary. Suppose that $\ell: \mathcal{N} \to S$ enumerates S, say $S := \{s_1, s_2, \ldots, s_n, \ldots\}$. Then we need only convert ℓ to a 1-1 enumeration. To do this we keep a list of the elements enumerated so far, $\langle \ell(0), \ldots, \ell(i) \rangle$ and define $f(n)$ to be the n-th distinct element in the enumeration of ℓ. We use the fact that S is discrete to test when n distinct elements have been enumerated. The following program does the job.

program	comments
J←1; K←0; A[0]←ℓ(0); M←0	
WHILE M<N DO	until N distinct elements are found, keep looking for a new one.
IF ∀K(0<K<M => ℓ(J)≠A[K])	Is ℓ(J) new.
THEN (M←M+1; A[M]←ℓ(J); J←J+1)	If yes make room for it, add it to A, get new inc
ELSE J←J+1	to look at a new element
FI	
END	
HALT(A[N])	Return the N-th distinct element.

To prove that the program is correct, we must produce a bound on J as a function of m. This bound comes from the proof that S is infinite. To prove that S is infinite we must have a rule ω which for each $m \in \mathcal{N}$ finds an $\omega(m) \in S$ with $\omega(m) > m$. Thus to prove that the program halts it is sufficient to consider the sequence

$$\omega(0),\ \omega(\omega(0)),\ldots,\ \omega^{(n)}(0).$$

The program will halt in less than $\omega^{(n)}(0)$ iterations. Notice, the bound $\omega^{(n)}(0)$ may be very crude so that the program stops considerably before that value. If it is a crude bound, then we have an imbalance between our information about the program and its behavior. This finishes (a).

The proof for part (b) is just as above but we must notice that the program can be expressed as a recursion equation iff $\lambda(\ )$ is a recursive function. Q.E.D.

Remark: Notice, if we had a recursive proof that S is infinite, then we could use an explicitly bounded program. Suppose that $\omega(\ )$ is the recursive bound mentioned in the proof. Then we could replace

UNTIL K = N DO

by

TO $\omega^{(N)}(0)$ UNTIL K=N DO.

We now arrive at one of the classical theorems about recursively enumerable sets.

Theorem 2 : If S is a computable, non-empty, proper subset of $\mathcal{N}$, then S is recursive iff S and $\bar{S}$ are recursively enumerable.

Proof: Clearly if S is recursive then to enumerate S recursively, use the program

```
(N): IF N ∈ S THEN HALT(N)
                ELSE HALT(a_0)
     FI
```

where $a_0 \in S$. To enumerate $\bar{S}$, replace $N \in S$ by $N \notin S$, and a_0 by some $b_0 \in \bar{S}$. By hypothesis we can find a_0 and b_0. Since S is recursive, the predicates "$N \in S$" and "$N \notin S$" are recursive, so then the enumerators are recursive also.

Conversely, suppose S and $\bar{S}$ are recursively enumerable by $\ell: \mathcal{N} \to S$ and $\bar{\ell}: \mathcal{N} \to \bar{S}$ respectively. Notice, since S is non-empty and proper, such functions exist.

The naive algorithm for deciding whether $n \in S$ is to enumerate S and $\bar{S}$ until n appears in one or the other. The algorithm is

```
     I←0
(N): UNTIL (V=N or W=N) DO
        (V←ℓ(I); W←ℓ̄(I); I←I+1)
     IF V=N THEN HALT(1)
           ELSE HALT(0)
     FI
```

To prove that this algorithm is correct, we must know that for all n there is some value of I for which the program halts. How do we know this? We must know that $n \in S$ or $n \in \bar{S}$. But we have a rule which will decide this, say rule α. Then once we know

whether $n \in S$ or $n \in \bar{S}$ we can determine I because we must have a proof that $\ell: \mathcal{N} \to \bar{S}$ is onto. This proof involves a function $\ell^{-1}: S \to \mathcal{N}$, such that $\ell(\ell^{-1}(s)) = s \in S$. Likewise there is a function $\bar{\ell}^{-1}: \bar{S} \to \mathcal{N}$. The bound on I is given by

$$I \leq (\text{if } \alpha(n) \text{ then } \ell^{-1}(n) \text{ else } \bar{\ell}^{-1}(n)) \qquad \text{Q.E.D.}$$

Remarks: Constructively this theorem appears uninteresting. We can reformulate it as:

If $\emptyset \neq S \subset \mathcal{N}$ and $S \cup \bar{S} = \mathcal{N}$ then
S is recursive iff S and $\bar{S}$ are r.e.

In this form we can see the classical mathematician's interest in it. He assumes for all subsets of $\mathcal{N}$ that $S \cup \bar{S} = \mathcal{N}$, thus for him this is an assertion about all sets.

Does the fact that this theorem is constructively valid only for computable sets detract from its value or its meaning or add to it? What do you think?

Notice, when we say that a set S is recursively enumerable, we really have two functions

$f: \mathcal{N} \to S$ the enumerator
$g: S \to \mathcal{N}$ the onto-verifier.

To say that S is r.e. we require a pair of facts

<f( ) recursive, g( ) computable>.

Suppose we require more precise information, say

$$\langle f(\ ) \text{ recursive}, g(\ ) \text{ recursive}\rangle$$

are such sets recursive? We shall see later that they are not.

(2.7) recursively enumerable but not recursive sets

Are there any r.e. sets which are not recursive? Are there any enumerable sets which are not computable? We know examples of non-computable and non-recursive sets. Recall:

$$H := \{i_\alpha \mid \alpha(i_\alpha) \text{ is undefined}\}$$
$$K := \{i \mid \phi_i(i) \text{ halts}\}.$$

We know that H is not computable (hence not recursive either), and we know K is not recursive (but possibly computable if Church's thesis is false).

We can show that K is recursively enumerable. The idea of proof is very interesting in its own right. It is related to one of the practical concepts in the organization of the "operating system" of a real computer, namely the concept of multi-processing.

Our plan to enumerate K is to try running every program on its own index, e.g. $\phi_0(0)$, $\phi_1(1)$, $\phi_2(2)$,... . Whenever a program halts, we enumerate its index. Eventually every program which halts will be enumerated (and we can tell exactly when if we know how many steps the program takes before halting).

In order to carry out this plan, we need a very important fact, perhaps the fundamental theorem of the subject. Namely, we must know that there is one program (or recursive procedure) which can execute any other program.

Theorem (Kleene-Turing): The set of all (n-argument) partial recursive functions, $\mathscr{PR}^n$, is recursively enumerable. That is, we can find a $\phi_{\mu_n} \in \mathscr{PR}^{n+1}$ such that

$$\phi_{\mu_n}(i,x_1,\ldots,x_n) \simeq \phi_i(x_1,\ldots,x_n) \text{ for all } i,x_1,\ldots,x_n.$$

Proof: We do not prove the theorem here. See [7] or [3] for clear proofs. The idea is essentially to build a universal program or recursive procedure, ϕ_{μ_n}.

Remark: There is no theorem of this type for all computing procedures. That is, we cannot at present prove that

* The set of all partial functions is enumerable.

It is unlikely that we shall ever prove this. One reason is that it is unlikely that we will ever believe we can enumerate all computing procedures. The best we can do is define a specific "closed set" of procedures and enumerate them. Any precisely defined programming language, such as Algol will be such a closed system.

Given the universal procedure ϕ_{μ_n}, we can elaborate on our plans for enumerating the programs (or procedures) ϕ_i which halt on their indices.

We imagine the list $\phi_0(0)$, $\phi_1(1)$, $\phi_2(2)$,... as an infinite queue of jobs waiting to be run on our control computer. We must devise a strategy for serving every job on the queue, regardless of how long it takes. We must also recognize that some jobs may not halt (who knows what a programmer will do).

Our method of satisfying each user is to put him on a service queue. Then we process every job on the service queue for Δ steps (called a service slice -- as opposed to American twist). If this service is enough to finish a job, we remove it from the service queue, and list the job as complete. If this slice of Δ steps is not enough to finish a job, then we leave it on the service queue. Before serving the finite ser-

vice queue again, we add a new job from the infinite queue. (In fact, we think of this infinite queue as being generated by a program called the ϕ-GENERATOR).

A procedure like this will be called DOVETAIL. We can think of it accepting any sequence, finite or infinite, of jobs. A job will be a program and an input to the program.

As this process executes, it generates a list. The list is K. A function DOVETAILLIST(i) picks out the i-th element on the list.

We could program the procedure described above, but that would be difficult because of the need to keep track of the data areas used by each program's partial execution. Since we are only trying to prove a theorem, not write an operating system, we will use a much simpler version of the idea. First we define a step counting measure.

Definition 1 : Given a recursive procedure ϕ_i, define a new procedure, denoted $t_0\phi_i$, called the step counting program, which satisfies the equation:

$$t_0\phi_i(x) := \begin{cases} \text{number of steps taken by algorithm } \phi_i \text{ on} \\ \text{input } x \text{ if } \phi_i(x) \text{ is defined,} \\ \text{undefined.} \qquad\qquad \text{otherwise} \end{cases}$$

Abbreviation: : Let $T_0(i,x,y)$ abbreviate the predicate $t_0\phi_i(x) \leq y$. The predicate is clearly recursive.

Proposition 1 : $\phi_i(x)$ halts iff $\exists y\ T_0(i,x,y)$.

Proof: For reader.

We now describe our simple version of DOVETAIL (or multi-processing). Instead of processing each ϕ_i for Δ steps and then restarting it again, we run ϕ_i for Δ steps, then starting over again later for 2Δ steps, then 3Δ steps, etc. The exact procedure for enumerating the halting indices is:

Definition 2: The program DOVETAIL LIST( ) is defined by

```
(N): K←0; M←0; OUT[0]←0;<----------------------we know that φ0(0)↓,
                                                by convention.
     UNTIL K=N DO
       BEGIN
         A[M]←M; B[M]←0<-----------------------initialize queue & t
                                                for all items on ser
         FOR J FROM 1 TO M DO                   queue do
            IF A[J] ≠ 0<-----------------------if J is still in que
            THEN IF t0φJ(J)≤B[J]<--------------see if φJ halts in n
                                                time     limit
                 THEN K←K+1: OUT[K]←J; A[J]←0<--if it halts remove i
                                                from queue (set A[J]
                 ELSE B[J]←B[J]+Δ<--------------if it does not halt,
                                                give it more steps f
                 FI                             next attempt
            ELSE←J←J+1<------------------------go to next index on
                                                service queue
            FI
         M←M+1<-------------------------------add a new index to
                                                service queue.
       END
     HALT(OUT[K])
```

We now want to prove that DOVETAILLIST( ) is a total function which enumerates K. It is easy to show that the function is total. We need only exhibit a subsequence of total programs, say $\phi_{s(i)}$, which halt in a constant number of steps. Then we can easily compute an upper bound on K for any input N. We leave this to the read

To show that DOVETAILLIST( ) enumerates K, we must show that it is onto K. Thus we need a function g( ) from K to $\mathcal{N}$.

The rule defining g: $K \to \mathcal{N}$ is

[On input x compute DOVETAILLIST(i), i = 0,1,..., until DOVETAILLIST(i_0) = x. Then output i_0.]

We must only prove g is a rule, that is:

* $\forall x(x \in K \Rightarrow \exists y\ \text{DOVETAILLIST}(y) = x)$

To prove * we must show that there is a procedure which given any x, constructs a proof of

$$x \in K \Rightarrow \exists y\ \text{DOVETAILLIST}(y) = x.$$

To construct a proof of a conditional we need only give a rule which will convert a proof of $x \in K$ into a proof of

** $\exists y\ \text{DOVETAILLIST}(y) = x$.

To prove $x \in K$ we must show that $\phi_x(x)$ is defined. By our previous convention that $\phi_x(x)$ is undefined iff it does not halt, we need only prove

$$\phi_x(x) \text{ halts.}$$

By Proposition 1 , to say $\phi_x(x)$ halts means that we can find an n such that

*** $t_0\phi_x(x) \leq n$.

Given the n of *** it is easy to define a number y for **.

To define a number we must give an expression which can be converted to a decimal numeral. The expression is a rule to determine this decimal. The rule is this.

> [First x appears in A[x] after at most x executions of "FOR J FROM 1 TO M." Once x appears on the service queue, A[J], it remains only until $t_0\alpha_x(x) \leq n$. Thus it remains there for only n/Δ executions of B[x] ← B[x] + Δ. This requires at most $O(x+n/\Delta)n/\Delta)$ steps (assuming no index leaves the queue and ignoring constants). So compute DOVETAILLIST(i) i = 0,1,... for at most $c\cdot((x+n/\Delta)\cdot n/\Delta)$ steps and determine the number y such that DOVETAILLIST(y)=x. Convert y to decimal.]

We have thus proved:

Theorem 1 : The set K is recursively enumerable.

The result suggests that we inquire whether H is enumerable. We soon see however that the proof that K is r.e. required the universal procedure, an algorithm not available for the class of all algorithms. Other considerations suggest that "H is enumerable" is unlikely. Therefore we list it as an unknown statement.

Question : Is H enumerable?

Do we have any examples of non-r.e. sets? Using Theorem 2 §2. we know.

Proposition 2: If $K \cup \bar{K} = \mathcal{N}$, then $\bar{K}$ is not r.e.

Proof: Since K is not recursive the result is immediate. Q.E.D.

Classically this result is interesting because the classical mathematician assumes $S \cup \bar{S} = \mathcal{N}$ for all subsets of $\mathcal{N}$. But can we find a more interesting constructively valid statement? We can in fact prove:

Theorem 2: $\bar{K}$ is not r.e.

Proof: Since we are proving a negation, $\neg$ ($\bar{K}$ r.e.), we can proceed by contradiction. Assume that $f: \mathcal{N} \to \bar{K}$ enumerates $\bar{K}$. We know that f is some recursive procedure, say ϕ_i. We can convert the range of f( ) into the domain of a partial function as follows:

$$\phi_{g(i)}(y) := (\text{if } \exists z\ f(z) = y \text{ then } 1 \text{ else } \Omega)$$

That is, g(i) is the index of the procedure

$$(Y):\ I \leftarrow 0;\ \text{UNTIL } Z = Y \text{ DO}(Z \leftarrow \phi_i(I);\ I \leftarrow I+1)$$

We know then that

$$y \in \bar{K} \text{ iff } y \in \text{range } \phi_i \text{ iff } y \in \text{domain } \phi_{g(i)}.$$

Now we claim that the index $g(i) \in \bar{K}$ but $g(i) \notin$ range of f. Since domain $\phi_{g(i)} \subseteq \bar{K}$ we know that if $x \in$ domain $\phi_{g(i)}$ then $\neg\ \phi_x(x)\downarrow$.

So we ask, does $\phi_{g(i)}(g(i))\downarrow$? If yes, then $g(i) \in \bar{K}$

which means $\phi_{g(i)}(g(i))\downarrow$. Therefore we know[†]

$\neg\ \phi_{g(i)}(g(i))\downarrow$. Thus $g(i) \notin$ range f.

This means that we know

$g(i) \in \bar{K}$ range(f)

Therefore f does not enumerate all of $\bar{K}$. Q.E.D.

The same proof can be used to show

Theorem 3 : $\bar{H}$ is not enumerable.

We shall examine some of the methods underlying this proof in §3. Now we consider some further simple facts about enumerable and recursively enumerable sets.

Theorem 4 :

(a) If S is an infinite enumerable subset of $\mathcal{N}$, then we can find a computable infinite subset of S.

(b) If S is a recursively enumerable infinite subset of $\mathcal{N}$, then we can find a recursive infinite subset of S.

To prove the theorem we use the following lemma. Both proofs are left to the reader.

[†]To be precise this step is: assume $\phi_{g(i)}(g(i)\downarrow)$, then we can deduce $\neg\ (\phi_{g(i)}(g(i))\downarrow)$. Thus we have A & $\neg$ A, which is a contradiction. Thus $\neg\ (\phi_{g(i)}(g(i))\downarrow)$.

Lemma 1 : If $f: \mathcal{N} \xrightarrow{\text{onto}} S$ is an enumerator of S and if f( ) is strictly monotonic ($f(x)<f(x+1)$), then S is computable.

(2.8) equivalent characterizations of enumerable and r.e. sets

We consider several general properties of enumerable and recursively enumerable sets.

Theorem 5 (Projection) :

(a) S is a non-empty enumerable proper subset of $\mathcal{N}$ iff it has the form $S = \{x|\exists y\ P(x,y)\}$ for P( ) a computable non-trivial predicate (i.e. $P(x_0,y_0)$ for some x_0,y_0 and $\neg P(x_1,y_1)$ some x_1,y_1).

(b) S is a non-empty recursively enumerable subset of $\mathcal{N}$ iff it has the form $S = \{x|\exists y\ P(x,y)\}$ for P( ) a recursive non-trivial predicate).

Proof: We prove only (a) leaving (b) for the reader.

If S is enumerable, then $f: \mathcal{N} \to S$ enumerates it. Define P(x,y) as $f(y) = x$, then clearly $S = \{x|\exists y\ P(x,y)\}$.

Conversely, given a predicate P( ), if it is non-trivial, then there is some pair such that $P(x_0,y_0)$. Thus x_0 will be in S. To enumerate S, just compute P(x,y) for all pairs $\langle x,y\rangle$ and whenever P(x,y) is true, enumerate x; otherwise enumerate x_0. Q.E.D.

We now characterize classes of sets closely related to the class of enumerable sets.

Definition 3 : Call S a domain set iff there is a procedure $\alpha: D \to D$ such that $S := \{x|\alpha(x)\downarrow\}$. Call S a partial recursive domain set (r.d.

set) iff there is a partial recursive procedure ϕ_i: D → D such that S:= $\{x|\phi_i(x)\downarrow\}$. The standard notation for partial recursive domain sets over $\mathcal{N}$ is w_i, thus

$$w_i := \{x|\phi_i(x)\downarrow\}.$$

We sometimes call r.d. sets, acceptable sets (especially in automata theory).

Definition 4 : Call R a range set iff there is a procedure f:D → D such that S:= $\{y|\exists x\ f(x) = y\}$. Call R a recursive range set (r.r. set iff there is a recursive procedure ϕ_i:D → D such that

$$S := \exists\{y|\exists x\ \phi_i(x) = y\}.$$

We sometimes call range sets partially enumerable sets and r.r. sets partially r.e. sets.

Theorem 6 :

(a) Every range subset of a discrete enumerable set is a domain set and conversely.

(b) Every recursive range subset of $\mathcal{N}$ is a recursive domain set and conversely.

Proof: To prove (a), we are given α: D → D and S:= range α.

Define the procedure β by:

[on input x, compute $\alpha(d_0)$, $\alpha(d_1)$,...,$\alpha(d_n)$,... until (if ever) α(y) = x, then halt and output 1]

where d_0, d_1,... is an enumeration of the domain D.

It is easy to prove

$$x \in S \text{ iff } \beta(x) = 1 \text{ iff } \beta(x)\downarrow.$$

Thus S is also a domain set determined by β.

Notice, this proof relies on the fact that the domain D is enumerable and discrete. Conversely, given $S = \{x \mid \beta(x)\downarrow\}$ define the rule α:

[on input x, compute $\beta(x)$; if this converges, output x].

Then $\alpha(x) = x$ iff $\beta(x)\downarrow$. Thus

$$x \in S \text{ iff } x \in \text{range } \alpha.$$

This finishes part (a). We leave part (b) for the reader. Q.E.D.

When $\alpha: D \overset{\text{onto}}{\rightarrow} S$ is also 1-1, then this is a theorem about inverting functions.

Corollary:

(a) If S is an enumerable subset of a discrete enumerable set, then S is a domain set.

(b) If S is an r.e. subset of $\mathcal{N}$, then S is a recursive domain set.

Proof: An enumerable set is a special type of range set.

To complete the set of relationships among domains, ranges, and enumerations, we mention

Theorem 7 :

(a) If S is a non-empty domain set of a discrete enumerable set U, then it is an enumerable set.

(b) If S is a non-empty domain set of $\mathcal{N}$, then it is an r.e. subset of $\mathcal{N}$.

§3 Proofs and Programs

(3.1) A critical appraisal of constructive logic

The classical and constructive views of the recursive functions, $\mathcal{R}$, are more different than the results of §2 suggest. These differences emerge in debates over the basic concepts. To illustrate them we will discuss a frequently proffered criticism of the constructive approach we have adopted here.

Classical Criticism

Constructive predicate logic of §1 clearly presupposes the concept of an effectively computable function, just as classical predicate logic presupposes the concept of truth. This state of affairs provokes the following points:

(1) One of the main accomplishments of modern logic was the definition of the effectively computable functions in terms of recursive functions (or equivalent formalisms such as Turing machines). Constructive logic as formulated here ignores this major achievement.

(2) To remedy the problem in (1), a rigorous constructivist must first introduce the concept of a recursive function. This requires the complete apparatus of Turing machines, or the like, in the foundations of logic. Such an apparatus is too complicated as a basis for mathematics.

(3) Moreover, to define effectively computable functions in terms of Turing machines or recursion equations requires already some concept of truth to allow us to say that "an algorithm is a procedure which halts on all inputs."

Let us unravel these objections. They are all based on a fundamental misunderstanding of the nature of the recursive functions.

First it is clear that before we can understand the concept of a recursive function or a Turing computable function, we must have some primitive concept of rule. Otherwise we would not know what it means to "substitute numeral $\underline{x}$ for variable y" or "move a tape head one cell left and print a one." In fact, as we have shown, this concept of rule and of computable function is as viable as the informal concept of set and can serve as a basis for mathematics in the same way that sets can. Thus objection (1) is false and with it falls objection (2).

The third objection is much more basic, but as easily answered. The classical mathematician will say that a recursion equation or Turing machine α is an algorithm iff it is <u>true</u> <u>that</u>

$$* \quad \forall x \, \exists! \, y(\alpha(x) = y).$$

But the only way that such truths are established mathematically is by proofs. Therefore if he wants a concrete and explicit definition of an algorithm, even a classical mathematician must have α and a proof of *.

Thus recursiveness is not a complete formal definition of an algorithm until it has been augmented with a formal definition of proof. Thus the classical concept of an algorithm must be extracted in a rather indirect manner from the classical concept of proof. On the other hand, the constructive notion of algorithm is made explicit by the axioms and rules of proof.

I would argue on the basis of these observations that the constructive method of defining effectively computable functions is simpler and more basic than the classical. Thus not only is

objection (3) specious along with (1) and (2), but the objections can all be turned around to read:

(1') One of the main accomplishments of constructive mathematics is the axiomatization of the concept of computability. Classical mathematics has chosen to ignore these insights.

(2') Furthermore, the classical notion of computability is erected on the complex notion of recursiveness which is interesting but philosophically unnecessary.

(3') The classical concept of computability is based on an implicit informal concept of rule. This concept is difficult to extract from the axioms.

(3.2) advantages of formalism

The computational meaning of many mathematical statements is useful in the applications of mathematics. This usefulness is enhanced manyfold when the compuation can be carried out by a machine. All such mechanical computations require that the mathematical statement be given formally. Consequently certain formal statements and proofs have a premium value.

At present the emphasis on producing machine executable versions of mathematical statements has placed a very large value on a certain level of statements, the imperative, and has devalued statements more concerned with the higher level meaning, such as assertions. This imbalance will be restored when the emphasis in applications returns to a need to thoroughly understand the con-

structions being used. The more abstract mathematical meaning is the level for human understanding. It has developed in that role for centuries. When that level is smoothly integrated into the lower levels, then we will have useful formal systems of mathematics, which include algorithmic languages as part of the underlying logic.

At the moment there are formal systems of mathematics lurking about in logic. But these are not useful in the same sense that Algol is and were never intended to be. They do not have an algorithmic component, and they are not used for doing mathematics, only for studying mathematical language. Such systems are not examples of the type we envision. But these systems and algorithmic languages taken together illustrate the value of formalism.

On one hand formalism is important because it leads to mechanical implementations. On the other hand, it is important because it provides a standard of precision which allows us to take such objects as proofs and rules as mathematical objects themselves, objects with the same precision as numbers. Thus at the formal level it is possible to imagine connections between formal computability and other mathematical problems.

One of the most dramatic illustrations of the purely intellectual value of formalism is the proof that Hilbert's 10th problem is recursively unsolvable. In computer science there are other deep examples of the extent of recursive unsolvability. For instance we know that equivalence of context free grammars is recursively unsolvable as is the problem of detecting ambiguity.

These specific examples of recursive unsolvability are interesting to some mathematicians only because they believe that recursive unsolvability means the same as absolute unsolvability. Such a stand does not necessarily appreciate these results. It is impressive enough, for my taste at least, that we know these problems are Algol unsolvable (which is equivalent to recursively unsolvable).

(3.3) Church's Thesis

There is evidence that the set of all Algol programs or all recursive procedures is sufficiently rich that any rule could be translated into such a formalism. One of the first people to realize this was the logician Alonzo Church who proposed a refinement of Constructivist Thesis II. He proposed in 1936 what is now known as Church's Thesis, namely that the concept of an effectively computable partial function be identified with the notion of a partial recursive function. Turing proposed a similar thesis in 1936 involving Turing computable functions.

From the classical point of view one speaks of evidence for the truth of this thesis. Such evidence is summarized in Kleene [7].

Constructively we do not speak of evidence because we can recognize the thesis as a matter of definition. We either choose to take this thesis as a new defining axiom about rules or we choose not to take it and rely instead on other axioms which appear more basic.

The question we ask about this axiom is whether it is based on overwhelming a priori intuition or whether it is based on more illusive metaphysical speculation. It appears that we can survive very well without the axiom. Its absence does not offend our intuition nor impair our mathematics. If we are interested in the consequences of the axiom, we can study them without comiting ourselves to its validity.

Choosing not to accept "Church's axiom," we are able to reflect certain aspects of classical mathematics in our theory (see Theorem 2 in §2.6). Thus in my view we lose an interesting accomodation with classical mathematics by adopting the axiom.

(3.4) conclusion

In conclusion let me summarize my reasons for suggesting a widespread use of a constructive mathematical language in computing theory. I will consider philosophical, methodological and technical reasons..

Philosophical Considerations

(1) I personally do not understand the meaning of many classical statements. For instance, I can imagine a constructive interpretation of the claim

* $\forall x$ ($\alpha_i(x)$ halts or $\alpha_i(x)$ does not halt)

for α_i an Algol program. But I can not imagine any constructive meaning to the claim

** For all functions f( ) and type 2 predicates P

$$P[f(\ ),x] \vee \neg P[f(\ ),x].$$

(2) I do not think that the classical definition of a terminating procedure (i.e. algorithm) is acceptable. It should be inherent in the definition that we know when the procedure will halt.

We could consider these philosophical objections more systematically as Bishop [1] and Constable [3] do, but space permits only these brief remarks.

Methodological Considerations

(3) Computer science is concerned with computation. If we use a constructive mathematical language, then the subject is self-contained and conceptually simpler than if we use a classical mathematical language.

(4) A constructive language keeps the scientist constantly aware of computational issues. Thus the language focuses our attention on the important questions and helps us exclude irrelevant issues.

(5) A constructive definition of the formal concept of an algorithm helps us avoid the fundamental misunderstanding about algorithms discussed in §3.1. Recently computer scientists such as Dijkstra have been compelled to speak at length to clear up problems arising from this misunderstanding.

(6) The constructive language is able to subsume the classical language in a meaningful way whereas the classical

language is only able to treat the constructive results as a separate domain. Some theorems in §2 have illustrated this (Thm §2.2, Thm §2.6 Thm 2). Thus the constructive language is richer than the classical.

Technical Considerations

(7) Formal systems of constructive mathematics provide excellent prototype models of new very high level programming languages. See the discussions in Constable [4] and Bishop [2].

(8) New theorems are suggested by the constructive viewpoint. A trivial illustration was given here in §2.

Remark. Other examples can be found in Constable [3].

(9) The constructive viewpoint allows us to consider the computational complexity of any function and thus extend efficiency considerations to all mathematical questions.

(10) There is an anomaly in the classical definition of the recursive functions. Namely, that definition requires certain ad hoc rules such as R1, R2 in §1. There is no justification for these rules. We can consider adding further rules R3, R4, ... each of which preserves the class of functions. But we can not "pass to the limit" and say that a function f( ) is recursive iff it is the unique solution to a system of equations. If that definition were used classically then we could define the function

$$h(n) := \begin{cases} \text{the least } y \text{ such that } t\alpha_n(n) \leq y\text{-}1 \\ \text{if such a } y \text{ exists; otherwise } 0. \end{cases}$$

I hope that the reader will be encouraged to try speaking a constructive language. Because our language shapes the world we see, learning the new language will open new worlds.

Acknowledgements

Many of the ideas expressed here arose out of discussions with Errett Bishop. The technical content has been helped by discussions with my students, especially John Cherniavsky, Kurt Mehlhorn and Steve Muchnick. The final form of the report has been enhanced by the conscientious typing of Sharon Gunkel and the proofreading of local readers (especially Steve Muchnick).

BIBLIOGRAPHY

[1] Bishop, E., Foundations of Constructive Analysis, McGraw Hill, N.Y., 1967, 370 pp.

[2] Bishop, E., "Mathematics as a Numerical Language", Intuitionism & Proof Theory, Myhill, J. et. al., North-Holland, Amsterdam, 1970, pp 53-71.

[3] Constable, R.L., Mathematical Computing Theory,vol I, Prentice-Hall, planned for 1974 printing.

[4] Constable, R.L., "Constructive Mathematics and Automatic Program Writers", Proceedings of IFIP Congress, Ljubljana, 1971.

[5] Fitting, M.C., Intuitionistic Logic, Model Theory and Forcing, North-Holland, Amsterdam, 1969, p. 191.

[6] Heyting, A., Intuitionism, North-Holland, Amsterdam, pp 145.

[7] Kleene, S.C., Introduction to Metamathematics, D. Van Nostrand, Princeton, 1952, 550 pp.

[8] Kleene, S.C. and R. E. Vesley, Foundations of Intuitionistic Mathematics, North-Holland, Amsterdam, 1965, 206 pp.

[9] Rogers, H.,Jr., Theory of Recursive Functions and Effective Computability, McGraw-Hill, N.Y., 1967, 482 pp.