Towards a Mathematical Science of Computation

J. McCarthy
Computer Science Department
Stanford University
Stanford, CA 95305
`jmc@cs.stanford.edu`
`http://www-formal.stanford.edu/jmc/`

1996
May 14, 1:20 p.m.

Abstract

1 Introduction

In this paper I shall discuss the prospects for a mathematical science of computation. In a mathematical science, it is possible to deduce from the basic assumptions, the important properties of the entities treated by the science. Thus, from Newton's law of gravitation and his laws of motion, one can deduce that the planetary orbits obey Kepler's laws.

What are the entities with which the science of computation deals?

What kinds of facts about these entities would we like to derive?

What are the basic assumptions from which we should start?

What important results have already been obtained? How can the mathematical science help in the solution of practical problems?

I would like to propose some partial answers to these questions. These partial answers suggest some problems for future work. First I shall give some very sketchy general answers to the questions. Then I shall present some recent results on three specific questions. Finally, I shall try to draw some conclusions about practical applications and problems for future work.

This paper should be considered together with my earlier paper [McC63]. The main results of the present paper are new but there is some overlap so that this paper would be self-contained. However some of the topics in this paper such as conditional expressions, recursive definition of functions, and proof by recursion induction are considered in much greater detail in the earlier paper.

2 What Are The Entities With Which Computer Science Deals?

These are problems, procedures, data spaces, programs representing procedures in particular programming languages, and computers.

Problems and procedures are often confused. A problem is defined by the criterion which determines whether a proposed solution is accepted. One can understand a problem completely without having any method of solution.

Procedures are usually built up from elementary procedures. What these elementary procedures may be, and how more complex procedures are constructed from them, is one of the first topics in computer science. This subject is not hard to understand since there is a precise notion of a computable function to guide us, and computability relative to a given collection of initial functions is easy to define.

Procedures operate on members of certain data spaces and produce members of other data spaces, using in general still other data spaces as intermediates. A number of operations are known for constructing new data spaces from simpler ones, but there is as yet no general theory of representable data spaces comparable to the theory of computable functions. Some results are given in [McC63].

Programs are symbolic expressions representing procedures. The same procedure may be represented by different programs in different programming languages. We shall discuss the problem of defining a programming language semantically by stating what procedures the programs represent. As for the syntax of programming languages, the rules which allow us to determine whether an expression belongs to the language have been formalized, but the parts of the syntax which relate closely to the semantics have not been so well studied. The problem of translating procedures from one programming language to another has been much studied, and we shall try to give a definition of the correctness of the translation.

Computers are finite automata. From our point of view, a computer is defined by the effect of executing a program with given input on the state of its memory and on its outputs. Computer science must study the various ways elements of data spaces are represented in the memory of the computer and how procedures are represented by computer programs. From this point of view, most of the current work on automata theory is beside the point.

3 What Kinds of Facts About Problems, Procedures, data Spaces, Programs, And Computers Would We Like to Derive?

Primarily. we would like to be able to prove that given procedures solve given problems. However, proving this may involve proving a whole host of other kinds of statement such as:

1. Two procedures are equivalent, i.e. compute the same function.

2. A number of computable functions satisfy a certain relationship, such as an algebraic identity or a formula of the functional calculus.

3. A certain procedure terminates for certain initial data, or for all initial data.

4. A certain translation procedure correctly translates procedures between one programming language and another.

5. One procedure is more efficient than another equivalent procedure in the sense of taking fewer steps or requiring less memory.

6. A certain transformation of programs preserves the function expressed but increases the efficiency.

7. A certain class of problems is unsolvable by any procedure, or requires procedures of a certain type for its solution.

4 What Are The Axioms And Rules of Inference of A Mathematical Science of Computation?

Ideally we would like a mathematical theory in which every true statement about procedures would have a proof, and preferably a proof that is easy to find, not too long, and easy to check. In 1931, Gödel proved a result, one of whose immediate consequences is that there is no complete mathematical theory of computation. Given any mathematical theory of computation there are true statements expressible in it which do not have proofs. Nevertheless, we can hope for a theory which is adequate for practical purposes, like proving that compilers work; the unprovable statements tend to be of a rather sweeping character, such as that the system itself is consistent.

It is almost possible to take over one of the systems of elementary number theory such as that given in Mostowski's book "Sentences Undecidable in Formalized Arithmetic" since the content of a theory of computation is quite similar. Unfortunately, this and similar systems were designed to make it easy to prove meta-theorems about the system, rather than to prove theorems in the system. As a result, the integers are given such a special role that the proofs of quite easy statements about simple procedures would be extremely long.

Therefore it is necessary to construct a new, though similar, theory in which neither the integers nor any other domain, (e.g. strings of symbols) are given a special role. Some partial results in this direction are described in this paper. Namely, an integer-free formalism for describing computations has been developed and shown to be adequate in the cases where it can be compared with other formalisms. Some methods of proof have been developed, but there is still a gap when it comes to methods of proving that a procedure terminates. The theory also requires extension in order to treat the properties of data spaces.

5 What Important Results Have Been Obtained Relevant to A Mathematical Science of Computation?

In 1936 the notion of a computable function was clarified by Turing, and he showed the existence of universal computers that, with an appropriate program, could compute anything computed by any other computer. All our stored program computers, when provided with unlimited auxiliary storage, are universal in Turing's sense. In some subconscious sense even the sales departments of computer manufacturers are aware of this, and they do not advertise magic instructions that cannot be simulated on competitors machines, but only that their machines are faster, cheaper, have more memory, or are easier to program.

The second major result was the existence of classes of unsolvable problems. This keeps all but the most ignorant of us out of certain Quixotic enterprises such as trying to invent a debugging procedure that can infallibly tell if a program being examined will get into a loop.

Later in this paper we shall discuss the relevance of the results of mathematical logic on creative sets to the problem of whether it is possible for a machine to be as intelligent as a human. In my opinion it is very important to build a firmer bridge between logic and recursive function theory on the one side, and the practice of computation on the other.

Much of the work on the theory of finite automata has been motivated by the hope of applying it to computation. I think this hope is mostly in vain because the fact of finiteness is used to show that the automaton will eventually repeat a state. However, anyone who waits for an IBM 7090 to repeat a state, solely because it is a finite automaton, is in for a very long wait.

6 How Can A Mathematical Science of Computation Help in The Solution of Practical Problems?

Naturally, the most important applications of a science cannot be foreseen when it is just beginning. However, the following applications can be foreseen.

1. At present, programming languages are constructed in a very unsystematic way. A number of proposed features are invented, and then we argue about whether each feature is worth its cost. A better understanding of the structure of computations and of data spaces will make it easier to see what features are really desirable.

2. It should be possible almost to eliminate debugging. Debugging is the testing of a program on cases one hopes are typical, until it seems to work. This hope is frequently vain.

Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. For this to be possible, formal systems are required in which it is easy to write proofs. There is a good prospect of doing this, because we can require the computer to do much more work in checking each step than a human is willing to do. Therefore, the steps can be bigger than with present formal systems. The prospects for this are discussed in [McC62].

7 Using Conditional Expressions to Define Functions Recursively

In ordinary mathematical language, there are certain tools for defining new functions in terms of old ones. These tools are composition and the identification of variables. As it happens, these tools are inadequate computable in terms of old ones. It is then customary to define all functions that can reasonably be regarded as to give a verbal definition. For example, the function $|x|$ is usually defined in words.

If we add a single formal tool, namely *conditional expressions* to our mathematical notation, and if we allow conditional expressions to be used recursively in a manner that I shall describe, we can define, in a completely formal way, all functions that can reasonably be regarded as computable in terms of an initial set. We will use the ALGOL 60 notation for conditional expressions in this paper.

We shall use conditional expressions in the simple form

$$\textbf{if } p \textbf{ then } a \textbf{ else } b$$

where p is a propositional expression whose value is **true** or **false**. The value of the conditional expression is a if p has the value **true** and b if p has the value **false**. When conditional expressions are used, the stock of predicates one has available for constructing p's is just as important as the stock of ordinary functions to be used in forming the a's and b's by composition. The following are examples of functions defined by conditional expressions:

1. $|x| = \textbf{if } x < 0 \textbf{ then } -x \textbf{ else } x$

2. $n! = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (n-1)!$

3. $n! = g(n, 1)$
   where
   $g(n, s) = \textbf{if } n = 0 \textbf{ then } s \textbf{ else } g(n-1, n \times s)$

4. $n! = f(n, 0, 1)$
   where
   $f(n, m, p) = \textbf{if } m = n \textbf{ then } p \textbf{ else } f(n, m+1, (m+1) \times p)$

5. $n^- = \mathit{pred}(n, 0)$
   where
   $\mathit{pred}(n, m) = \textbf{if } m' = n \textbf{ then } m \textbf{ else } \mathit{pred}(n, m')$

6. $m + n = \textbf{if } n = 0 \textbf{ then } m \textbf{ else } m' + n^-$

The first of these is the only non-recursive definition in the group. Next, we have three different procedures for computing $n!$; they can be shown to be equivalent by the methods to be described in this paper. Then we define the predecessor function n^- for positive integers ($3^- = 2$) in terms of the successor function $n'(2' = 3)$ Finally, we define addition in terms of the successor, the predecessor and equality. In all of the definitions, except for the first, the domain of the variables is taken to be the set of non-negative integers.

As an example of the use of these definitions, we shall show how to compute 2! by the second definition of $n!$. We have

$$\begin{aligned}
2! &= g(2, 1) \\
&= \textbf{if } 2 = 0 \textbf{ then } 1 \textbf{ else } g(2-1, 2 \times 1) \\
&= g(1, 2) \\
&= \textbf{if } 1 = 0 \textbf{ then } 2 \textbf{ else } g(1-1, 1 \times 2) \\
&= g(0, 2) \\
&= \textbf{if } 0 = 0 \textbf{ then } 2 \textbf{ else } g(0-1, 0 \times 2) \\
&= 2
\end{aligned}$$

Note that if we attempted to compute $n!$ for any n but a non-negative integer the successive reductions would not terminate. In such cases we say that the computation does not converge. Some recursive functions converge for all values of their arguments, some converge for some values of the arguments, and some never converge. Functions that always converge are called *total* functions, and the others are called *partial* functions. One of the earliest major results of recursive function theory is that there is no formalism that gives all the computable total functions and no partial functions.

We have proved [McC63] that the above method of defining computable functions, starting from just the successor function n' and equality, yields all the computable functions of integers. This leads us to assert that we have the complete set of tools for defining functions which are computable in terms of given base functions.

If we examine the next to last line of our computation of 2! we see that we cannot simply regard the conditional expression

$$\textbf{if } p \textbf{ then } a \textbf{ else } b$$

as a function of the three quantities p, a, and b. If we did so, we would be obliged to compute $g(-1, 0)$ before evaluating the conditional expression, and this computation would not converge. What must happen is that when p is true we take a as the value of the conditional expression without even looking at b.

Any reference to recursive functions in the rest of this paper refers to functions defined by the above methods.

8 Proving Statements About Recursive Functions

In the previous section we presented a formalism for describing functions which are computable in terms of given base functions. We would like to have a mathematical theory strong enough to admit proofs of those facts about computing procedures that one ordinarily needs to show that computer programs meet their specifications. In [McC63] we showed that our formalism for expressing computable functions, was strong enough so that all the partial recursive functions of integers could be obtained from the successor function and equality. Now we would like a theory strong enough so that the addition of some form of Peano's axioms would allow the proof of all the theorems of one of the forms of elementary number theory.

We do not yet have such a theory. The difficulty is to keep the axioms and rules of inference of the theory free of the integers or other special domain, just as we have succeeded in doing with the formalism for constructing functions.

We shall now list the tools we have so far for proving relations between recursive functions. They are discussed in more detail in [McC63].

1. Substitution of expressions for free variables in equations and other relations.

2. Replacement of a sub-expression in a relation by an expression which has been proved equal to the sub-expression. (This is known in elementary mathematics as substitution of equals for equals.) When we are dealing with conditional expressions, a stronger form of this rule than the usual one is valid. Suppose we have an expression of the form

$$\mathbf{if}\ p\ \mathbf{then}\ a\ \mathbf{else}\ (\mathbf{if}\ q\ \mathbf{then}\ b\ \mathbf{else}\ c)$$

and we wish to replace b by d. Under these circumstances we do not have to prove the equation $b = d$ in general, but only the weaker statement

$$\sim p \wedge q \supset b = d$$

This is because b affects the value of the conditional expression only in case $\sim p \wedge q$ is true.

3. Conditional expressions satisfy a number of identities, and a complete theory of conditional expressions, very similar to propositional calculus, is thoroughly treated in [McC63]. We shall list a few of the identities taken as axioms here.

1. $(\mathbf{if\ true\ then}\ a\ \mathbf{else}\ b) = a$

2. $(\mathbf{if\ false\ then}\ a\ \mathbf{else}\ b) = b$

3. $\mathbf{if}\ p\ \mathbf{then}\ (\mathbf{if}\ q\ \mathbf{then}\ a\ \mathbf{else}\ b)\ \mathbf{else}\ (\mathbf{if}\ q\ \mathbf{then}\ C\ \mathbf{else}\ d) =$
   $\mathbf{if}\ q\ \mathbf{then}\ (\mathbf{if}\ p\ \mathbf{then}\ a\ \mathbf{else}\ c)\ \mathbf{else}\ (\mathbf{if}\ p\ \mathbf{then}\ b\ \mathbf{else}\ d)$

4. $f(\mathbf{if}\ p\ \mathbf{then}\ a\ \mathbf{else}\ b) = \ \mathbf{if}\ p\ \mathbf{then}\ f(a)\ \mathbf{else}\ f(b)$

4. Finally, we have a rule called *recursion induction* for making arguments that in the usual formulations are made by mathematical induction. This rule may be regarded as taking one of the theorems of recursive function theory and giving it the status of a rule of inference. Recursion induction may be described as follows:

Suppose we have a recursion equation defining a function f, namely

$$f(x_1, \ldots, x_n) = \varepsilon\{f, x_1, \ldots, x_n\} \tag{1}$$

where the right hand side will be a conditional expression in all non-trivial cases, and suppose that

1. the calculation of $f(x_1, \ldots, x_n)$ according to this rule converges for a certain set A of n-tuples,

2. the functions $g(x_1, \ldots, x_n)$ and $h(x_1, \ldots, x_n)$ each satisfy equation (1) when g or h is substituted for f. Then we may conclude that $g(x_1, \ldots, x_n) = h(x_1, \ldots, x_n)$ for all n-tuples $(x_1, \ldots, x_n)$ in the set A.

As an example of the use of recursion induction we shall prove that the function g defined by

$$g(n, s) = \textbf{if } n = 0 \textbf{ then } s \textbf{ else } g(n-1, n \times s)$$

satisfies $g(n, s) = n! \times s$ given the usual facts about $n!$. We shall take for granted the fact that $g(n, s)$ converges for all non-negative integral n and s. Then we need only show that $n! \times s$ satisfies the equation for $g(n, s)$. We have

$$\begin{aligned} n! \times s \quad = \quad & \textbf{if } n = 0 \textbf{ then } n! \times s \textbf{ else } n! \times s \\ & \textbf{if } n = 0 \textbf{ then } s \textbf{ else } (n-1)! \times (n \times s) \end{aligned}$$

and this has the same form as the equation satisfied by $g(n, s)$. The steps in this derivation are fairly obvious but also follow from the above-mentioned axioms and rules of inference. In particular, the extended rule of replacement is used. A number of additional examples of proof by recursion induction are given in [McC63], and it has been used to prove some fairly complicated relations between computable functions.

9 Recursive Functions, Flow Charts, And Algolic Programs

In this section 1 want to establish a relation between the use of recursive functions to define computations, the flow chart notation, and programs expressed as sequences of ALGOL-type assignment statements, together with conditional **go to**'s. The latter we shall call *Algolic* programs with the idea of later extending the notation and methods of proof to cover more of the language of ALGOL. Remember that our purpose here is not to create yet another programming language, but to give tools for proving facts about programs.

In order to regard programs as recursive functions, we shall define the *state vector* ξ of a program at a given time, to be the set of current assignments of values to the variables of the program. In the case of a machine language program, the state vector is the set of current contents of those registers whose contents change during the course of execution of the program.

When a block of program having a single entrance and exit is executed, the effect of this execution on the state vector may be described by a function $\xi' = r(\xi)$ giving the new state vector in terms of the old. Consider a program described by the flow chart of fig. 1.

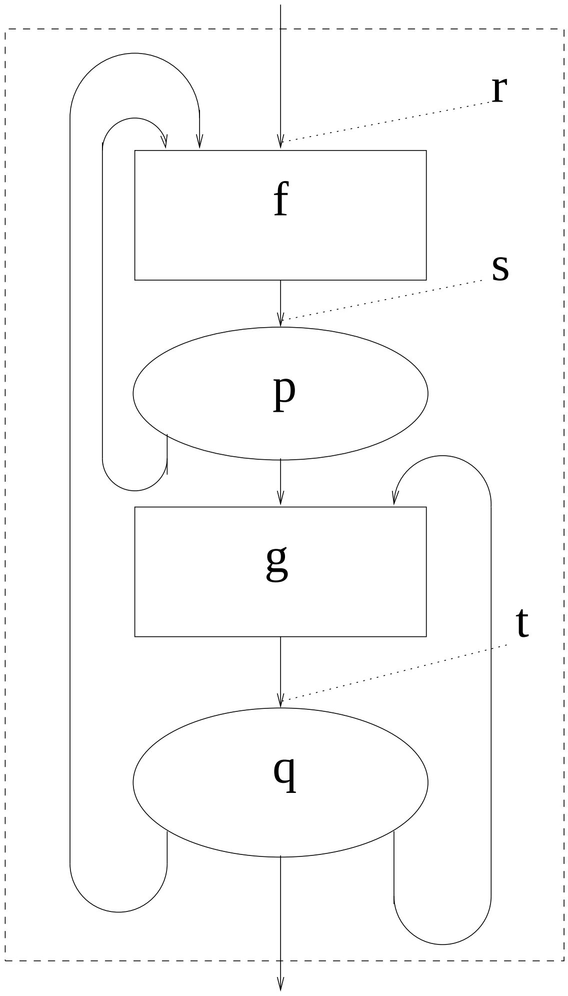

Fig. 1 Flow chart for equations shown below.

$$
\begin{aligned}
r(\xi) &= s(f(\xi)) \\
s(\xi) &= \textbf{if } p(\xi) \textbf{ then } r(\xi) \textbf{ else } t(g(\xi)) \\
t(\xi) &= \textbf{if } q_1(\xi) \textbf{ then } r(\xi) \textbf{ else if } q_2(\xi)\ \xi \textbf{ else } t(g(\xi))
\end{aligned}
$$

The two inner computation blocks are described by functions $f(\xi)$ and $g(\xi)$, telling how these blocks affect the state vector. The decision ovals are described by predicates $p(\xi)$, $q_1(\xi)$, and $q_2(\xi)$ that give the conditions for taking the various paths. We wish to describe the function $r(\xi)$ that gives the effect of the whole block, in terms of the functions f and g and the predicates

p, q_1, and q_2. This is done with the help of two auxiliary functions s and t. $s(\xi)$ describes the change in ξ between the point labelled s in the chart and the final exit from the block; t is defined similarly with respect to the point labelled t. We can read the following equations directly from the flow chart:

$$\begin{aligned} r(\xi) &= s(f(\xi)) \\ s(\xi) &= \textbf{if } p(\xi) \textbf{ then } r(\xi) \textbf{ else } t(g(\xi)) \\ t(\xi) &= \textbf{if } q_1(\xi) \textbf{ then } r(\xi) \textbf{ else if } q_2(\xi) \textbf{ then } \xi \textbf{ else } t(g(\xi)) \end{aligned}$$

In fact, these equations contain exactly the same information as does the flow chart.

Consider the function mentioned earlier, given by

$$g(n, s) = \textbf{if } n = 0 \textbf{ then } s \textbf{ else } g(n - 1, n \times s)$$

It corresponds, as we leave the reader to convince himself, to the Algolic program

$$\begin{array}{ll} a: & \textbf{if } n = 0 \textbf{ then go to } b; \\ & s := n \times s; \\ & n := n - 1; \\ & \textbf{go to } a; \\ b: & \end{array}$$

Its flow chart is given in fig. 2.

This flow chart is described by the function $fac(\xi)$ and the equation

$$fac(\xi) = \textbf{if } p(\xi) \textbf{ then } \xi \textbf{ else } fac(g(f(\xi)))$$

where $p(\xi)$ corresponds to the condition $n = 0$, $f(\xi)$ to the statement $n := n - 1$, and $g(\xi)$ to the statement $s := n \times s$.

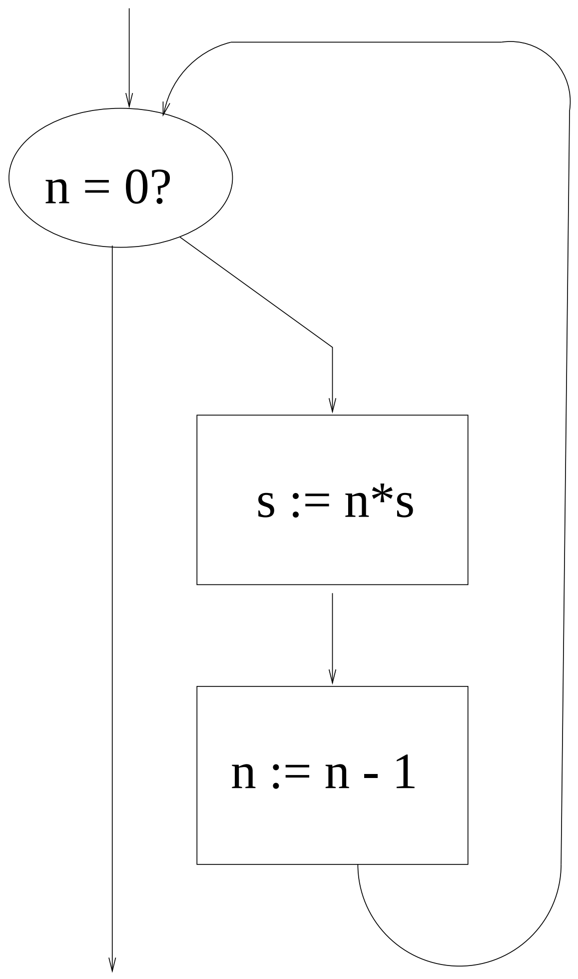

Fig. 2 Flow chart for $fac(\xi)$ as described by the following equations.

$$fac(\xi) = \textbf{if } p(\xi) \textbf{ then } \xi \textbf{ else } fac(g(f(\xi)))$$

where

$$\begin{aligned}
p(\xi) &\equiv c(n,\xi) = 0, \\
f(\xi) &= a(s, c(n,\xi) * c(s,\xi), \xi), \\
g(\xi) &= a(n, c(n,\xi) - 1, \xi)
\end{aligned}$$

and so

$$fac(\xi) = a(n, 0, a(s, c(n, \xi)! \times c(s, \xi))).$$

We shall regard the state vector as having many components, but the only components affected by the present program are the s-component and the n-component. In order to compute explicitly with the state vectors, we introduce the function $c(var, \xi)$ which denotes the value assigned to the variable var in the state vector ξ, and we also introduce the function $a(var, val, \xi)$ which denotes the state vector that results when the value assigned to var in ξ is changed to val, and the values of the other variables are left unchanged.

The predicates and functions of state vectors mentioned above then become:

$$\begin{aligned} p(\xi) &= (c(n, \xi) = 0) \\ g(\xi) &= a(n, c(n, \xi) - 1, \xi) \\ f(\xi) &= a(s, c(n, \xi) \times c(s, \xi), \xi) \end{aligned}$$

We can prove by recursion induction that

$$fac(\xi) = a(n, 0, a(s, c(n, \xi)! \times c(s, \xi), \xi),$$

but in order to do so we must use the following facts about the basic state vector functions:

1. $c(u, a(\mathrm{v}, \alpha, \xi)) = \mathbf{if}\ u = \mathrm{v}\ \mathbf{then}\ \alpha\ \mathbf{else}\ c(u, \xi)$

2. $a(\mathrm{v}, c(\mathrm{v}, \xi), \xi) = \xi$

3. $a(u, \alpha, a(\mathrm{v}, \beta, \xi)) = \mathbf{if}\ u = \mathrm{v}\ \mathbf{then}\ a(u, \alpha, \xi)\ \mathbf{else}\ a(\mathrm{v}, \beta, a(u, \alpha, \xi))$

The proof parallels the previous proof that

$$g(n, s) = n! \times s,$$

but the formulae are longer.

While all flow charts correspond immediately to recursive functions of the state vector, the converse is not the case. The translation from recursive function to flow chart and hence to Algolic program is immediate, only if the recursion equations are in *iterative* form. Suppose we have a recursion equation

$$r(x_1, \ldots, x_n) = \mathcal{E}\{r, x_1, \ldots, x_n, f_1, \ldots, f_m\}$$

where $\mathcal{E}\{r, x_1, = ldots, x_n, f_1, \ldots, f_m\}$ is a conditional expression defining the function r in terms of the functions $f_1, \ldots, f_m$. $\mathcal{E}$ is said to be in iterative form if r never occurs as the argument of a function but only in terms of the main conditional expression in the form

$$\ldots \textbf{ then } r(\ldots).$$

In the examples, all the recursive definitions except

$$n! = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (n-1)!$$

are in interative form. In that one, $(n-1)!$ occurs as a term of a product. Non-iterative recursive functions translate into Algolic programs that use themselves recursively as procedures. Numerical calculations can usually be transformed into iterative form, but computations with symbolic expressions usually cannot, unless quantities playing the role of push-down lists are introduced explicitly.

10 Recursion Induction on Algolic Programs

In this section we extend the principle of recursion induction so that it can be applied directly to Algolic programs without first transforming them to recursive functions. Consider the Algolic program

$$\begin{array}{ll} a: & \textbf{if } n = 0 \textbf{ then go to } b; \\ & s := n \times s; \\ & n := n - 1; \\ & \textbf{go to } a; \\ b: & \end{array}$$

We want to prove it equivalent to the program

$$\begin{array}{ll} a: & s := n! \times s \\ & n := 0 \\ b: & \end{array}$$

Before giving this proof we shall describe the principle of recursion induction as it applies to a simple class of flow charts. Suppose we have a block of program represented by fig. 3a. Suppose we have proved that this block is equivalent to the flow chart of fig. 3b where the shaded block is the original block again. This corresponds to the idea of a function satisfying a functional equation. We may now conclude that the block of fig. 3a is equivalent to the flow chart of fig. 3c for those state vectors for which the program does not get stuck in a loop in fig. 3c.

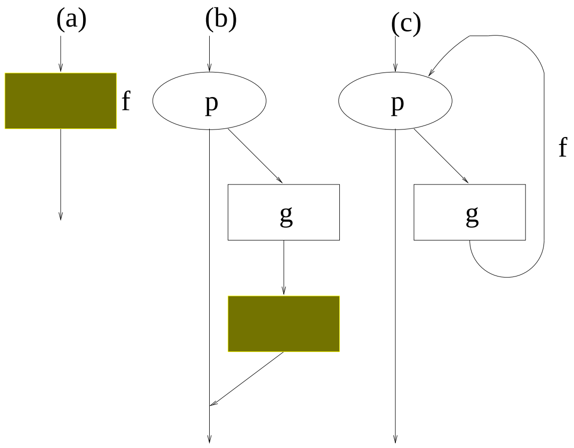

Fig. 3 Recursion induction applied to flow charts.

This can be seen as follows. Suppose that, for a particular input, the program of fig. 3c goes around the loop n times before it comes out. Consider the flow chart that results when the whole of fig. 3b is substituted for the shaded (or yellow) block, and then substituted into this figure again, etc. for a total of n substitutions. The state vector in going through this flow chart will undergo exactly the same tests and changes as it does in going through fig. 3c, and therefore will come out in n steps without ever going through the shaded (or yellow) block. Therefore, it must come out with the same value as in fig. 3c. Therefore, for any vectors for which the computation of fig. 3c converges, fig. 3a is equivalent to fig. 3c.

Thus, in order to use this principle to prove the equivalence of two blocks we prove that they satisfy the same relation, of type 3a-3b, and that the corresponding 3c converges.

We shall apply this method to showing that the two programs mentioned above are equivalent. The first program may be transformed as follows:

from

$$\begin{aligned} a: \quad & \textbf{if } n = 0 \textbf{ then go to } b; s := n \times s; n := n - 1; \\ & \textbf{go to } a : b : \end{aligned}$$

to

$$\begin{aligned} a: \quad & \textbf{if } n = 0 \textbf{ then go to } b; s := n \times s; n := n - 1; \\ & a_1 : \textbf{if } n = 0 \textbf{ then go to } b; s := n \times s; n := n - 1; \\ & \textbf{go to } a_1; b : \end{aligned}$$

The operation used to transform the program is simply to write separately the first execution of a loop.

In the case of the second program we go:

from

$$a : s := n! \times s; n := 0; b :$$

to

$$\begin{aligned} a: \quad & \textbf{if } n = 0 \textbf{ then go to } c; s := n! \times s; n := 0; \\ & \textbf{go to } b; c : s := n! \times s; n := 0; b : \end{aligned}$$

to

$$\begin{aligned} a: \quad & \textbf{if } n = 0 \textbf{ then go to } c; s := (n-1)! \times (n \times s); n := 0; \\ & \textbf{go to } b; c :; b : \end{aligned}$$

to

$$\begin{aligned} a: \quad & \textbf{if } n = 0 \textbf{ then go to } b; s := n \times s; n := n - 1; s := \\ & = n! \times s; n := 0; b : \end{aligned}$$

The operations used here are: first, the introduction of a spurious branch, after which the same action is performed regardless of which way the branch goes; second, a simplification based on the fact that if the branch goes one way then $n = 0$, together with rewriting one of the right-hand sides of an assignment statement; and third, elimination of a label corresponding to a null statement, and what might be called an anti-simplification of a sequence of assignment statements. We have not yet worked out a set of formal rules justifying these steps, but the rules can be obtained from the rules given above for substitution, replacement and manipulation of conditional expressions.

The result of these transformations is to show that the two programs each satisfy a relation of the form: *program* is equivalent to

$$a : \textbf{if } n = 0 \textbf{ then go to } b; s := n \times s; n := 0;\ \textit{program}.$$

The program corresponding to fig. 3c in this case, is precisely the first program which we shall assume always converges. This completes the proof.

11 The Description of Programming Languages

Programming languages must be described syntactically and semantically. Here is the distinction. The syntactic description includes:

1. A description of the *morphology*, namely what symbolic expressions represent grammatical programs.

2. The rules for the analysis of a program into parts and its synthesis from the parts. Thus we must be able to recognize a term representing a sum and to extract from it the terms representing the summands.

The semantic description of the language must tell what the programs mean. The meaning of a program is its effect on the state vector in the case of a machine independent language, and its effect on the contents of memory in the case of a machine language program.

12 Abstract Syntax of Programming Languages

The Backus normal form that is used in the ALGOL report, describes the morphology of ALGOL programs in a *synthetic* manner. Namely, it describes how the various kinds of program are built up from their parts. This would be better for translating into ALGOL than it is for the more usual problem of translating from ALGOL. The form of syntax we shall now describe differs from the Backus normal form in two ways. First, it is analytic rather than synthetic; it tells how to take a program apart, rather than how to put it together. Second, it is abstract in that it is independent of the notation used to represent, say sums, but only affirms that they can be recognized and taken apart.

Consider a fragment of the ALGOL arithmetic terms, including constants and variables and sums and products. As a further simplification we shall consider sums and products as having exactly two arguments, although the general case is not much more difficult. Suppose we have a term t. From the point of view of translating it we must first determine whether it represents a constant, a variable, a sum, or a product. Therefore we postulate the existence of four predicates: $isconst(t)$, $isvar(t)$, $issum(t)$ and $isprod(t)$. We shall suppose that each term satisfies exactly one of these predicates.

Consider the terms that are variables. A translator will have to be able to tell whether two symbols are occurrences of the same variable, so we need a predicate of equality on the space of variables. If the variables will have to be sorted an ordering relation is also required.

Consider the sums. Our translator must be able to pick out the summands, and therefore we postulate the existence of functions $addend(t)$ and $augend(t)$ defined on those terms that satisfy $issum(t)$. Similarly, we have the functions $mplier(t)$ and $mpcand(t)$ defined on the products. Since the analysis of a term must terminate, the recursively defined predicate

$$\begin{aligned} &term(t) = isvar(t) \vee isconst(t) \vee issum(t) \wedge term \\ &\quad (addend(t)) \wedge term(augend(t)) \vee isprod(t) \wedge term \\ &\quad (mplier(t)) \wedge term(mpcand(t)) \end{aligned}$$

must converge and have the value **true** for all terms.

The predicates and functions whose existence and relations define the syntax, are precisely those needed to translate from the language, or to define the semantics. That is why we need not care whether sums are represented by $a + b$, or $+ab$, or (PLUS A B), or even by Gödel numbers $7^a 11^b$.

It is useful to consider languages which have both an analytic and a synthetic syntax satisfying certain relations. Continuing our example of terms, the synthetic syntax is defined by two functions $mksum(t, u)$ and $mkprod(t, u)$ which, when given two terms t and u, yield their sum and product respectively. We shall call the syntax *regular* if the analytic and the synthetic syntax are related by the plausible relations:

1. $issum(mksum(t, u))$ and $isprod(mkprod(t, u))$
2. $addend(mksum(t, u)) = t; mplier(mkprod(t, u)) = t$
   $augend(mksum(t, u)) = u, mpcand(mkprod(t, u)) = u$

3. $issum(t) \supset mksum(addend(t), augend(t)) = t$ and
   $isprod(t) \supset mkprod(mplier(t), mpcand(t)) = t$

Once the abstract syntax of a language has been decided, then one can choose the domain of symbolic expressions to be used. Then one can define the syntactic functions explicitly and satisfy one self, preferably by proving it, that they satisfy the proper relations. If both the analytic and synthetic functions are given, we do not have the difficult and sometimes unsolvable analysis problems that arise when languages are described synthetically only.

In ALGOL the relations between the analytic and the synthetic functions are not quite regular, namely the relations hold only up to an equivalence. Thus, redundant parentheses are allowed, etc.

13 Semantics

The analytic syntactic functions can be used to define the semantics of a programming language. In order to define the meaning of the arithmetic terms described above, we need two more functions of a semantic nature, one analytic and one synthetic. If the term t is a constant then $val(t)$ is the number which t denotes. If α is a number $mkconst(\alpha)$ is the symbolic expression denoting this number. We have the obvious relations

1. $val(mkconst(\alpha)) = \alpha$
2. $isconst(mkconst(\alpha))$
3. $isconst(t) \supset mkconst(val(t)) = t$

Now we can define the meaning of terms by saying that the value of a term for a state vector ξ is given by

$$\begin{aligned} &value(t, \xi) = \mathbf{if}\ isvar(t)\ \mathbf{then}\ c(t, \xi)\ \mathbf{else\ if}\ isconst(t) \\ &\quad \mathbf{then}\ val(t)\ \mathbf{else\ if}\ issum(t)\ \mathbf{then}\ value\ (addend(t), \xi) + \\ &\quad + value(augend(t), \xi)\ \mathbf{else\ if}\ isprod(t)\ \mathbf{then}\ value(mplier(t), \\ &\quad \xi) \times value(mpcand(t), \xi) \end{aligned}$$

We can go farther and describe the meaning of a program in a programming language as follows: *The meaning of a program is defined by its effect* *on the state vector*. Admittedly, this definition will have to be elaborated to take input and output into account.

In the case of ALGOL we should have a function

$$\xi' = algol(\pi, \xi),$$

which gives the value ξ' of the state vector after the ALGOL program π has stopped, given that it was started at its beginning and that the state vector was initially ξ. We expect to publish elsewhere a recursive description of the meaning of a small subset of ALGOL.

Translation rules can also be described formally. Namely,

1. A machine is described by a function

$$x' = machine(p, x)$$

giving the effect of operating the machine program p on a machine vector x.

2. An invertible representation $x = rep(\xi)$ of a state vector as a machine vector is given.

3. A function $p = trans(\pi)$ translating source programs into machine programs is given.

The correctness of the translation is defined by the equation

$$rep(algol(\pi, \xi)) = machine(trans(\pi), rep(\xi)).$$

It should be noted that $trans(\pi)$ is an abstract function and not a machine program. In order to describe compilers, we need another abstract invertible function, $u = repp(\pi)$, which gives a representation of the source program in the machine memory ready to be translated. A *compiler* is then a machine program such that $trans(\pi) = machine(compiler, repp(\pi))$.

14 The Limitations of Machines And Men as Problem-Solvers

Can a man make a computer program that is as intelligent as he is? The question has two parts. First, we ask whether it is possible in principle, in view of the mathematical results on undecidability and incompleteness. The second part is a question of the state of the programming art as it concerns artificial intelligence. Even if the answer to the first question is "no", one can still try to go as far as possible in solving problems by machine.

My guess is that there is no such limitation in principle. However, a complete discussion involves the deeper parts of recursive function theory, and the necessary mathematics has not all been developed.

Consider the problem of deciding whether a sentence of the lower predicate calculus is a theorem. Many problems of actual mathematical or scientific interest can be formulated in this form, and this problem has been proved equivalent to many other problems including problem of determining whether a program on a computer with unlimited memory will ever stop. According to Post, this is equivalent to deciding whether an integer is a member of what Post calls a *creative* set. It was proved by Myhill that all creative sets are equivalent in a quite strong sense, so that there is really only one class of problems at this level of unsolvability.

Concerning this problem the following facts are known:

1) There is a procedure which will do the following: If the number is in the creative set, the procedure will say "yes", and if the number is not in the creative set the procedure will not say "yes", it may say "no" or it may run on indefinitely.

2) There is no procedure which will always say "yes" when the answer is "yes", and will always say "no" when the answer is "no" If a procedure has property (1) it must sometimes run on indefinitely. Thus there is no procedure which can always decide definitely whether a number is a member of a creative set, or equivalently, whether a sentence of the lower predicate calculus is a theorem, or whether an ALGOL program with given input will ever stop. This is the sense in which these problems are unsolvable.

3) Now we come to Post's surprising result. We might try to do as well as possible. Namely, we can try to find a procedure that always says "yes" when the answer is "yes", and never says "yes" when the answer is "no", and which says "no" for as large a class of the negative cases as possible, thus narrowing down the set of cases where the procedure goes on indefinitely as much as we can. *Post showed that one cannot even do as well as possible.* Namely, Post gave a procedure which, when given any other partial decision procedure, would give a better one. The better procedure decides all the cases the first procedure decided, and an infinite class of new ones. At first sight this seems to suggest that Emil Post was more intelligent than any possible machine, although Post himself modestly refrained from drawing this conclusion. Whatever program you propose, Post can do better. However, this is unfair to the programs. Namely, Post's improvement procedure is itself mechanical and can be carried out by machine, so that the machine can

improve its own procedure or can even improve Post, if given a description of Post's own methods of trying to decide sentences of the lower predicate calculus. It is like a contest to see who can name the largest number. The fact that I can add one to any number you give, does not prove that I am better than you are.

4) However, the situation is more complicated than this. Obviously, the improvement process may be carried out any finite number of times and a little thought shows that the process can be carried out an infinite number of times. Namely, let p be the original procedure, and let Post's improvement procedure be called P, then $P^n p$ represents the original procedure improved n times. We can define $P^\omega p$ as a procedure that applies p for a while, then P_D for a while, then p again, then P_p, then $P^2 p$, then p, then P_p, then $P^2 p$, then $P^3 p$, etc. This procedure will decide any problem that any P^n will decide, and so may justifiably be called $P^\omega p$. However, $P^\omega p$ is itself subject to Post's original improvement process and the result may be called $P^{\omega+1} p$. How far can this go? The answer is technical. Namely, given any recursive transfinite ordinal α, one can define $P^\alpha p$. A recursive ordinal is a recursive ordering of the integers that is a well-ordering in the sense that any subset of the integers has a least member in the sense of the ordering. Thus, we have a contest in trying to name the largest recursive ordinal. Here we seem to be stuck, because the limit of the recursive ordinals is not recursive. However, this does not exclude the possibility that there is a different improvement process Q, such that Qp is better than $P^\alpha p$ for any recursive ordinal α.

5) There is yet another complication. Suppose someone names what he claims is a large recursive ordinal. We, or our program, can name a larger one by adding one, but how do we know that the procedure that he claims is a recursive well-ordering, really is? He says he has proved it in some system of arithmetic. In the first place we may not be confident that his system of arithmetic is correct or even consistent. But even if the system is correct, by Gödel's theorem it is incomplete. In particular, there are recursive ordinals that cannot be proved to be so in that system. Rosenbloom, in his book "Elements of Mathematical Logic" drew the conclusion that man was in principle superior to machines because, given any formal system of arithmetic, he could give a stronger system containing true and provable sentences that could not be proved in the original system. What Rosenbloom missed, presumably for sentimental reasons, is that the improvement procedure is itself mechanical, and subject to iteration through the recursive ordinals, so that we are back in the large ordinal race. Again we face the complication

of proving that our large ordinals are really ordinals.

6) These considerations have little practical significance in their present form. Namely, the original improvement process is such that the improved process is likely to be too slow to get results in a reasonable time, whether carried out by man or by machine. It may be possible, however, to put this hierarchy in some usable form.

In conclusion, let me re-assert that the question of whether there are limitations in principle of what problems man can make machines solve for him as compared to his own ability to solve problems, really is a technical question in recursive function theory.

References

[McC62] John McCarthy. Checking mathematical proofs by computer. In *Proceedings Symposium on Recursive Function Theory (1961).* American Mathematical Society, 1962.

[McC63] J. McCarthy. A basis for a mathematical theory of computation.[1] In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

/@steam.stanford.edu:/u/jmc/s96/towards.tex: begun 1996
May 1, latexed 1996

May 14 at 1:20 p.m.

[1] http://www-formal.stanford.edu/jmc/basis/basis.html