A programming approach to some concepts and results in the theory of computation

J. M. Brady

Computing Centre, University of Essex, Wivenhoe Park, Colchester, Essex.

The benefits of a programming approach to the theory of computation are illustrated by considering a traditional proof, namely the equivalence of Turing's formalism and general recursive functions (GRF). In these terms, we are led to criticise the GRF and to give a programmer's view of the Kleene normal form theorem.

(Received December 1973, revised May 1974)

1. Introduction

In this paper, we want to illustrate the benefits of a programming approach to the theory of computation. There are, we contend, two main such benefits:

1. As a pedagogical vehicle. Programming provides a natural framework in which to present concepts and results from the theory of computation to the general computing community. A similar assumption seems to underlie the recent papers by Hoare and Allison (1972) on incomputability, and Brinch-Hansen (1972) on operating systems.
2. As a research tool. Viewing the proof of a theorem as a program to solve an appropriate problem (for example) can illuminate the main trends in the proof, as well as suggesting a way to devise better proofs by better programs and perhaps better languages or systems. This is illustrated in Section 4.

Our central argument derives from the fact that the theory of computation is the attempt to build a theory or science out of computing phenomena, which in turn largely consists of writing programs to (try to) solve problems. Consequently, proof methodology should reflect problem solving methodology for problems attacked in computer science. Such a problem solving methodology should be embodied in the linguistic structures of the various programming languages in use by computer scientists. We suggest that one of the major reasons for the current shortcomings in the theory of computation, say with regard to proving programs correct, is that the proofs and proof structures are hardly ever related to the program structures. An exception to this is Hoare's work on proving the correctness of programs (Hoare, 1969; 1971; 1972; 1973; Hoare and Foley, 1972).

To illustrate the advantages of a programming approach, we take one of the most fundamental notions in the theory of computation. This arises from the realisation that to ask the basic question 'Is there a limit to what we can compute?' we must first decide precisely what is meant by 'computable'. As is well known, suggested precise notions of 'computable' began to appear in the 1930's; many have since been published. There seem to be two main approaches, which we may call the 'abstract machine' approach, and the 'functional' or 'programming' approach. The former approach, which includes the work of Turing (1936), Wang (1957), Shepherdson and Sturgis (1963), Elgot and Robinson (1964) and Minsky (1967), consists of developing mathematical models of executing mechanisms (processors) on which computations may be run. The latter approach, described more fully in Section 3, includes the work of Church (1941), Herbrand-Gödel-Kleene (see Hermes, 1965, ch. 5), Hilbert-Kleene (the so-called General Recursive functions) (see Hermes, 1965, ch. 3) and McCarthy (1963), and consists of developing essentially mathematical systems for 'programming' in. More precisely, such functional approaches define the semantics of a family of programming languages.

The earliest abstract machine approach was that devised by Turing. His proposal that Turing machines embody a precise notion of 'computable' is usually referred to as Turing's thesis: the things which one wants to think of as algorithmic, that is programmable, are precisely the things which can be computed by some Turing machine (hereafter TM). Observe that such a claim is not provable—we may only become convinced of it through argument. Turing's paper contains a delightful appeal to your intuition development of his abstract machine model, designed to constitute such a defence of his claim. Perhaps a more compelling argument is the fact that to date all suggested precise formulations of 'computable' have come up with an equivalent notion: for example, something is a general recursive function (GRF) precisely when it is computable according to Turing. What does this mean in programming terms? On the one hand we have the linguistic approach GRF, on the other the machine approach TM. We have proofs (see for example (Hermes, 1965, sections 16 and 18) that each GRF has a corresponding TM, and vice versa. Clearly this suggests a pair of programs: a TM simulator in GRF, and a compiler from GRF to TM. The main features of such a TM simulator in GRF are sketched and analysed in Section 4. In particular, it is shown how such a program brings out shortcomings in a system (GRF) apparently intended to be part of a theory of computation. We should emphasis that such a simulator does not constitute a *proof* that TMs and GRFs are equally powerful, because we will not attempt to prove our programs correct. That is not the point of the paper. Rather, it should be clear how a pair of proofs could be developed from the programs.

2. Turing machines

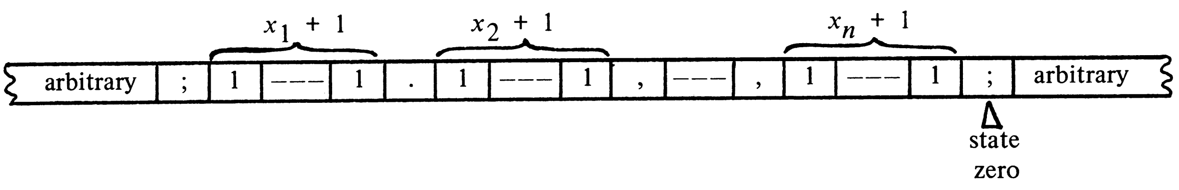
There are numerous highly readable (as well as varied) accounts of TMs in the literature, particularly (Minsky, 1967, chap. 6, 7). We here briefly review the main points to fix the discussion in Section 4. A TM (notionally) has a tape marked in squares together with a read/write head, and is capable of moving the head left/right one square, sensing a character in the square under the head, and altering the sensed character. The reading head exists in one of a finite number of states, one of which is 'halt'. The tape is of unlimited length (or can be made so by splicing) and is blank apart from a finite portion whose squares each contain a character from a finite character set C. The ability to move in either direction on the unlimited tape provides a facility of arbitrarily long memory since a written square can be revisited later. The execution cycle consists of altering the sensed character, the reading position and reading head state in a way which depends only on the sensed character and reading head state. That is, the machine executes:*

*The following program fragment, indeed all such included in the paper, is intended to communicate an algorithm to the reader, and is written in the language BCPL (Richards, 1970), since not only is it highly readable but we later need function-returning functions.

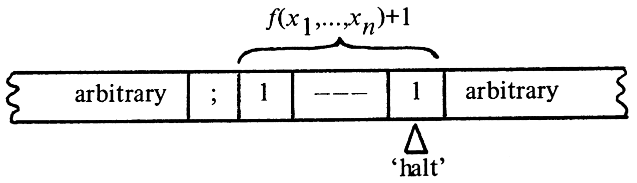
```
until state = 'halt' do
  $( let oldstate = state
    state := statefn (oldstate, character)
    scannedsquare := scannedsquare +
      (directionfn(oldstate, character) = right → 1, −1)*
    character := charfn(oldstate, character)
  $)
```

Since the purpose of TMs is to provide a precise and unambiguous definition of 'computable', we need to specify even the 'housekeeping' details in a definition, lest they be misinterpreted.

Definition
A function $f: N^n \rightarrow N$ (where N is the set of natural numbers) is Turing-computable if there is a TM m_f such that if m_f is supplied with the tape



and m_f eventually halts, it does so with the tape configured as follows



Notice that we have specified a unary-plus-one coding for numbers; this is to distinguish zero, coded as 1, from a blank square.

3. The functional or programming approach; GRF

The idea underlying this approach, which probably (see Hermes, 1965, paras. 13,4) originated with David Hilbert, is that a computable function or program is a constructed object; that is, has to be built.

The force of this idea is that if a computable function f is to be regarded as *built*, then the components $f_1, \ldots, f_m$ from which f is built are also computable functions, combined into f according to one of a number of what we shall call 'strategies'. Of course, it is reasonable to suppose that the component functions $f_1, \ldots, f_m$ have themselves to be built. If the components of f_i are $f_{i1}, \ldots, f_{in_i}$, $1 \leqslant i \leqslant m$, we may illustrate a preliminary analysis of f as in **Fig. 1.**

It seems reasonable to suppose that it takes (or took) only a finite amount of time, thought or effort to build any function. This means that the tree structure in Fig. 1 can only branch finitely at any point, according to one of a finite number of strategies, and can only be of finite depth. In particular there must be some functions which are *not* decomposable; we call these the *base* functions, and suppose they form a set F. If we assume the strategies form a finite set C, we can denote the totality of computable functions constructed by this approach by $C(F)$†.

If we view a programming language in these terms, the strategies C are the statement structuring mechanisms according to which statements are built out of their components, for example, the FORTRAN DO loop or ALGOL block, **for** or **if** statements. On the other hand, any functional approach defines the semantics of a whole family of programming languages. The first programming language with this as an explicit design criterion was LISP (McCarthy *et al*, 1962).

The general recursive functions (GRF) originated with Hilbert in the 1920s, and, understandably, manipulate only natural numbers. The set FH of base functions in GRF show the influence of Peano's axiomatisation of arithmetic and are:

1. The constant zero function c_0, defined by $c_0(n) = 0$,
2. The successor function $\text{succ}(n) = n + 1$,
3. The predecessor function pred, defined by
$$\text{pred}(n) = (n > 0 \rightarrow n - 1, 0)$$
4. A whole family of argument-selector functions p_{ni}, $n \geqslant 1$, $1 \leqslant i \leqslant n$, where
$$p_{ni}(x_1, \ldots, x_n) = x_i \ .$$

Hilbert originally proposed a set CH of two strategies (defined below) namely composition and recursion. This system $CH(FH)$ is now called the primitive recursive functions system, as Ackermann in 1928 built a program (see Hermes, 1965, para. 13) which could not be defined in Hilbert's system. Later Kleene added the minimisation operation. Hoare and Allison (1972) have characterised a subset of ALGOL 60 which computes primitive recursive functions.

Composition provides a means of sequencing by function nesting, and is the main form of sequencing used in LISP. If we have functions (subroutines) $g_1, \ldots, g_n$ taking the same number m of arguments (formal parameters) and a function (routine) f of n arguments (formal parameters), we can build a function (routine) h out of f, $g_1, \ldots, g_n$ by:
$$h(x_1, \ldots, x_m) = f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$
(To see how this provides sequencing, consider the evaluation of an application of h: first evaluate the subexpressions $g_1, \ldots, g_n$ and then apply f to their results). Instead of always having to invent a possibly spurious name h for each f, $g_1, \ldots, g_n$, we have a default or standard name Compos $(f, g_1, \ldots, g_n)$. This remark is similar to the use of λ-notation to give a standard name which describes its effects as a function.

The recursion strategy of GRF is unfortunately named, as its normal iterative use is more in the spirit of a FORTRAN DO or ALGOL **for** loop than recursion in ALGOL or LISP. The strategy is based on the observation that the principle of mathematical induction can equally well be used to construct functions as to prove theorems (see Manna and Waldinger, 1971 and Boyer and Strother-Moore, 1973 for more on this point). If we have g, h appropriately defined, we can build f by

$f(\mathbf{x}, y) = \textbf{test}\ y = 0\ \textbf{then}\ g(\mathbf{x})$	base of induction
$\textbf{or}\ h(\mathbf{x}, y, f(\mathbf{x}, y - 1))$	inductive step

where $\mathbf{x}$ stands for $x_1, \ldots, x_n$. A standard name for recursion, analogous to Compos, is Recn(g, h).

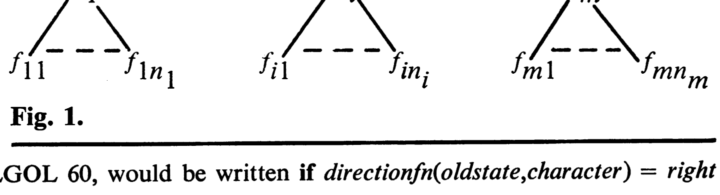


Fig. 1.

*This is BCPL notation for the conditional expression, which, in ALGOL 60, would be written **if** *directionfn(oldstate,character)* = *right* **then** 1 **else** −1.
†This notation is McCarthy's (1963).

The number of steps required to complete the evaluation of any primitive recursive function can be predicted from its form and its arguments. No primitive recursive function can generate computation such as a **while** command

The minimisation operator Min (called μ in the literature) of Kleene gets over this limitation. A version of it can be 'defined' by

```
Min(f, k) (x) = valof $( let y = 0
                    while f(x, y) ≠ k do y := y + 1
                        resultis y
                  $)
```

for example, if factimes$(x, y) = x*y!$, we have Min(factimes, 24) applied to 4 yields 3 since $4*3! = 24$ but Min(factimes, 24) applied to 3 is undefined since there is no number y such that $y! = 8$.

The primitive recursive functions together with Min define the general recursive functions.

4. Simulating Turing machines as GRF

We aim to sketch the construction of a simulator, so that if *tm* is a Turing machine, simulate(*tm*) is the GRF computed by *tm*. We want to program simulate so that the evaluation of simulate $(tm)(x_1, \ldots, x_n)$ exactly simulates the execution of *tm* when placed over a tape initially containing $x_1, \ldots, x_n$. Now the action of *tm* can readily be described as follows: after the initial tape is set up with $x_1, \ldots, x_n$ on it and the state initialised and reading head positioned, the execution cycle of *tm* is repeated until a halt state is reached. Finally, the result of the computation is extracted from the final tape. A consideration of the 'housekeeping' details built into the definition of Turing-computability in Section 2 shows that the process of initialising the tape and extracting the result are essentially independent of *tm*. Thus the 'top level' structure of our simulator is described by

Simulate$(tm)(x_1, \ldots, x_n) =$
$\qquad extractresult(mainloop(tm,initialise(x_1, \ldots, x_n)))$

Initialise, like all GRF functions, has to return a single number, which clearly is a coded version of all the information needed by *mainloop*: scanned square, scanned character, state and tape. There are several obvious ways to achieve this coding (see for example Minsky, 1967, 13.3.3 and 17.3). For the remainder of this section we shall suppose that we have devised a coding system which *initialise* can use to pass all relevant information, coded as a single integer and called an 'information packet', to *mainloop*. In these terms, the job of *mainloop* is to transform, according to *tm*, the information packet $initialise(x_1, \ldots, x_n)$ into an information packet representing the end of the computation. We suppose that we have selector functions *sel-state*, *sel-scanned-char*, etc. which allow us to access (decode) the state, scanned character, and tape description from the current information packet.

The evaluation of *mainloop(tm,infpack)* is to simulate the execution of *tm* starting on *infpack*, and can be described by

```
mainloop(tm,infpack) = valof $( while sel-state(infpack) ≠ halt
                              do
                                 infpack := next(tm, infpack);
                                 resultis infpack
                              $)
```

in which *next(tm,infpack)*, defined more precisely below, is the information packet resulting from one execution cycle of *tm* starting at *infpack*. This little program has the same basic format, namely a **while** statement, as that defining the Min strategy in Section 3, so it seems reasonable to suppose that a simple application of that strategy will complete the programming of *mainloop*. Unfortunately this is *not* the case, for closer inspection of the **while** statement of the Min strategy reveals that it takes a very restricted form, which only allows us to increment the control variable y at each step: in *mainloop* on the other hand we want to perform a quite complex operation *infpack* := *next(tm,infpack)*, corresponding to an execution cycle of *tm*. However we are going to have to involve Min somehow in the programming of *mainloop* because we cannot in general predict how many execution cycles *tm* will perform before halting. So we have to use Min with a control variable, say l, which is incremented by one for each execution cycle. *Such a control variable l clearly counts the number of execution cycles of tm.* The solution then is to use Min to program a subroutine *timesround* which returns the length l of the computation of *tm* on $x_1, \ldots, x_n$ and then use this *computed* or *a priori given* l together with another subroutine *transform* to program *mainloop* by Recn and Compos. That is,

Mainloop(tm,infpack) =
$\qquad$ *transform(tm,infpack,timesround(tm,infpack))*

Notice the use of function nesting, as described in Section 3, to achieve sequencing of *timesround* followed by *transform*. We can thus program *mainloop* as follows:

```
let timesround(tm,infpack) = ?   programmed using Min
and transform(tm,infpack,  = ?   programmed without using
  runtime)                       Min
let mainloop = Compos(transform, p21, p22, timesround)
```

where p_{21}, p_{22} are argument-selector functions, which select the first and second arguments, *tm* and *infpack* respectively, of *mainloop*.

The crucial point to note here is the following: *the linguistic inadequacy of the Min strategy, in particular, the restricted form of the embodied* **while** *statement, presented us with a programming problem, which in turn led to an intricate programming solution.* Although this is fairly transparent here, it is *not* so in most *mathematical* proofs that the GRFs are Turing-computable, for example Hermes (1965) and Minsky (1967, chapter 10). In particular, a modified Min strategy Min′, embodying a more general **while** statement, and defined by

```
Min′(f, k, g, yinit)(x) = valof $( let y = yinit
                          while f(x, y) ≠ k do
                                y := g(x, y)
                                resultis y
                                $)
```

leads to the following straightforward program (and proof).

```
        mainloop(tm,infpack) = ml(tm,infpack)(tm)
where
      ml(tm,infpack) = min′(sel-state,halt,next,infpack)
```

As it is, we are left to program *timesround* and *transform*. In fact, *timesround* can be programmed as Min(*getstate*,*halt*) where *getstate(tm,infpack,l)* is the state of *tm* after l cycles starting from *infpack*. In fact, as it is easy to see,

```
getstate(tm,infpack,l) = test l = 0 then sel-state(infpack)
                         or getstate(tm,next(tm,infpack),l−1)
```

This is not directly in the form for an application of Recn, revealing yet another linguistic inadequacy; in this case, Minsky (1967, 10.3.2) has devised the following simple technique to program around the problem.

Suppose f is defined by

$f(\mathbf{x}, y, l) =$ **test** $l = 0$ **then** $g(\mathbf{x}, y)$
$\qquad$ **or** $f(\mathbf{x}, h(\mathbf{x}, y), l - 1)$

where g and h are GRF. We can prove that f itself is a GRF as follows: Observe that

$f(\mathbf{x}, y, 0) = g(\mathbf{x}, y)$
$f(\mathbf{x}, y, 1) = f(\mathbf{x}, h(\mathbf{x}, y), 0) = g(\mathbf{x}, h(\mathbf{x}, y))$
$f(\mathbf{x}, y, 2) = f(\mathbf{x}, h(\mathbf{x}, y), 1) = g(\mathbf{x}, h(\mathbf{x}, h(\mathbf{x}, y)))$

and, generally for $l \geqslant 0$,

$f(\mathbf{x}, y, l) = g(\mathbf{x}, h(\mathbf{x}, h(\mathbf{x}, \ldots, h(\mathbf{x}, y) \ldots)$

there being l 'h's. If we define

$H(\mathbf{x}, y, l) =$ **test** $l = 0$ **then** y
$\qquad$ **or** $h(\mathbf{x}, H(\mathbf{x}, y, l - 1))$

then H is clearly GRF, and since

$H(\mathbf{x}, y, l) = h(\mathbf{x}, h(\mathbf{x}, \ldots, h(\mathbf{x}, y) \ldots)$

(with l 'h's),

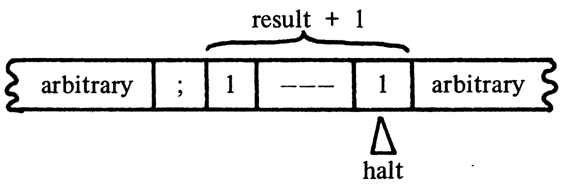
$f(\mathbf{x}, y, l) = g(\mathbf{x}, H(\mathbf{x}, y, l))$

so that f is GRF. An application of the same technique disposes of *transform*, leaving only *next*.

Now next(*tm*,*infpack*) is the information packet resulting from one execution cycle of *tm* starting at *infpack*. This means that we have to access the coded description *tm* to get the next state, next character and next direction. We have not so far discussed the coding of TMs, but it is fairly straightforward. Since the current state and scanned character completely determine the next state, character and direction, we can suppose we have a lookup routine such that lookup(*tm*,*state*,*char*) is a coded triple containing the next state, next character and next direction. We can also suppose we have a routine alter such that alter(*infpack*,*triple*) returns the updated *infpack* according to triple. Then

```
next(tm,infpack) =
  valof $( let triple = lookup(tm,sel-state(infpack),
              sel-char(infpack))
           resultis alter(infpack,triple)
        $)
```

This completes the programming of *mainloop*, and leaves only *extractresult*. Recalling the definition of Turing-computable, we note that the *infpack* returned by *mainloop* represents the final configuration:



The 'obvious' way to program *extractresult* is by a **while** or Min loop, moving left until a semicolon is reached. It turns out that this is *not* in fact necessary, because the coding system used by *initialise* codes the entire tape contents as a single number. Since the tape is only finitely inscribed, this number is finite and can provide an upper bound to the computation of *extractresult*. Consequently we can introduce a bounded minimum operator *Bdmin*, program *Bdmin* in terms of Recn and Compos and program *extractresult* in terms of *Bdmin*, In fact, it is much easier to define *Bdmin* in English as follows:

$$\textit{Bdmin}(f, k)(\mathbf{x}, l) = \textit{the least } y \textit{ such that } 0 \leqslant y \leqslant l \textit{ and } f(\mathbf{x}, y) = k, \textit{ or zero if there is no such } y.$$

than to program in BCPL:

```
Bdmin(f, k)(x, l) =
    (l = 0 → 0,
     Bdmin(f, k)(x, l − 1) = 0 ∧ f(x, 0) ≠ k
              (f(x, l) = k → l, 0),
     Bdmin(f, k)(x, l − 1))
```

It turns out (Hermes, 1965, para. 18) that if g is primitive recursive, so is *Bdmin*(g, k), so that we really do not need Min to program *extractresult*, although as may be readily seen it is clearer and probably more efficient to do so in practice. Gathering together all our programming effort we find that only a single application of the Min strategy (to build *timesround* out of *getstate*) is needed to program the GRF which simulates *tm*. Since any GRF is computable by some *TM*, this means that *any* GRF can be programmed with only a single application of Min. This result (that any GRF can be obtained by a single application of Min to some primitive recursive function) is usually called the Kleene normal form theorem, and implies that it is theoretically possible to write any ALGOL program (for example) with only a single **for** . . . **while** statement.

5. Conclusion

Viewing the problem of proving that any GRF is Turing-computable as a programming problem, led us to criticise the **while** mechanism of the GRF, and to suggest a more general alternative, which made for easier programming; that is, a simpler proof.

Acknowledgement

The author thanks the referee for his many helpful suggestions.

References

Boyer, R., and Strother-Moore, J. (1973). *Proving theorems about LISP functions*, Department Computation Logic memo 60, Edinburgh.

Brinch-Hansen, P. (1972). An outline of a course on Operating Systems Principles, in *Operating Systems Techniques* (ed. Hoare and Perrott), Academic Press.

Church, A. (1941). The Calculi of λ-conversion, *Annals of Maths Studies*, 6, Princeton.

Elgot, C. C., and Robinson, A. (1964). Random-Access Stored-Program Machines: An approach to programming languages. *JACM*, Vol. 11, No. 4, pp. 356-399.

Hermes, H. (1965). *Enumerability, decidability, computability*, Springer.

Hoare, C. A. R., and Allison, D. C. S. (1972). Incomputability, *ACM Computing Surveys*, Vol. 4, No. 3, pp. 169-178.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming, *CACM*, Vol. 12, No. 10, pp. 576-583.

Hoare, C. A. R. (1971). Proof of a program: FIND, *CACM*, Vol. 14, No. 1, pp. 39-45.

Hoare, C. A. R. (1973). Proof of a structured program: 'The sieve of Erastosthenes'. *The Computer Journal*, Vol. 15, pp. 321-325.

Hoare, C. A. R. (1972). Proof of correctness of data representation, *Acta Informatica*, Vol. 1, pp. 271-281.

Hoare, C. A. R., and Foley, M. (1972). Proof of a recursive program; Quicksort, *The Computer Journal*, Vol. 14, pp. 391-395.

Manna, Z., and Waldinger, R. J. (1971). Towards automatic program synthesis, *CACM*, Vol. 14, No. 3, pp. 151-165.

McCarthy, J. (1963). A basis for a mathematical theory of computation, in *Computer Programming and Formal Systems* (Braffort and Hirshberg eds.), North-Holland, p. 33-69.

McCarthy, J. *et al.*(1962). *The LISP 1.5 programmer's manual*, MIT press.

Minsky, M. L. (1967). *Computation: finite and infinite machines*, Prentice-Hall.

Richards, M. (1970). *The BCPL reference manual*, University of Essex.

Shepherdson, J. C., and Sturgis, H. E. (1963). Computability of recursive functions, *JACM*, Vol. 10, pp. 217-255.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungs problem, *Proc. Lond. Math. Soc.*, Ser. 2-42, pp. 230-265.

Wang, H. (1957). A variant to Turing's theory of Computing machines, *JACM*, Vol. 4, No. 1.