Hints on proofs by recursion induction*

J. M. Brady

Computing Centre, University of Essex, Wivenhoe Park, Colchester CO4 3SQ, Essex

In 1963 John McCarthy proposed a formalism based on conditional expressions and recursion for use in the emergent theory of computation. Included in his proposals was a proof technique, known as recrusive induction, which could be used to establish the equivalence of recursively defined functions. This paper shows that the discovery of an equation to serve in a proof by recursive induction does not have to rely on luck or inspiration, but can be developed rationally hand in hand with the development of the proof.

(Received November 1975)

In 1963, John McCarthy (1963) published a seminal paper in which he proposed a formalism based on conditional expressions and recursion for use in the emergent theory of computation. He also introduced a class of nonnumerical symbolic expressions which he called S-expressions, and conjectured that there might be a calculus of S-expressions analogous to number theory. To this end, one of his more important contributions was a proof technique known as *recursion induction*, which could be used to establish the equivalence of recursively defined functions, including functions of S-expressions. Suppose that $g(x)$ and $h(x)$ are recursively defined functions which we want to prove equivalent; for example we might have $g(x) =$ reverse [append $[x1, x2]$] and $h(x) =$ append [reverse $[x_2]$, reverse $[x_1]$]. Recursion induction asserts that we may conclude that g is equivalent to h if we can find an *equation*

$$f(x) = \varepsilon(x, f)$$

which we can prove (a) defines a function f and (b) is satisfied by g and h when they are substituted for f. Morris (1971) points out that the justification of this claim is based on Kleene's first recursion theorem (Kleene 1950, page 348).

Despite its intended application to functions of S-expressions, we illustrate the method with a simple application to the natural numbers due to McCarthy (1963, page 59). Define addition in terms of the successor x^+ and predecessor x^- as follows:

$$x + y = (x = 0 \rightarrow y, x^- + y^+) \ . \qquad (1)$$

We can prove that $(x + y)^+ = x + (y^+)$ as follows:
let $g(x, y) = (x + y)^+$ and $h(x, y) = x + (y^+)$, and use the equation

$$f(x, y) = (x = 0 \rightarrow y^+, f(x^-, y^+)) \ . \qquad (2)$$

A simple induction proof on x establishes that (2) defines a function, so we are left to show that it is satisfied by g and h.

$$\begin{aligned} g(x, y) &= (x + y)^+ \\ &= (x = 0 \rightarrow y, x^- + y^+)^+ && \text{//definition of addition} \\ &= (x = 0 \rightarrow y^+, (x^- + y^+)^+) && \text{//property of conditional†} \\ &= (x = 0 \rightarrow y^+, g(x^-, y^+)) && \text{//definition of } g \end{aligned}$$

so that g satisfies (2). The proof for h is similar.

In the above proof, we have followed the normal dictates of mathematical elegance and given no clue as to the process by which f was discovered. Other instances of such elegance in the theory of computation include the discovery of Ackermann's function and the Grzegrocyck hierarchy. (Elsewhere we have shown how the process of discovering them can quite easily be explained (Brady, 1977).)

Experience soon shows that once an appropriate equation such as (2) has been discovered, the proof usually proceeds fairly straightforwardly; the most difficult problem is seen to be the *discovery* of the equation for f. This paper discusses that discovery process. In particular, we argue that a major misconception is that the discovery of the equation for f is independent of, and precedes, the proof of equivalence. On the contrary, we shall show how the equation may be discovered rationally, if heuristically, by matching (appropriate expansions of) the equations for g and h and previously proved theorems. It transpires that such matches only succeed when there is a certain structural similarity in the equations roughly corresponding to a similarity in the control structures embodied in the interpretation of the recursive definitions. For example, it is very difficult to prove that $(x^+) + y = (x + y)^+$ without changing the definition of addition to

$$(x + y) = (y = 0 \rightarrow x, x^+ + y^-) \qquad (3)$$

(in which case one can use a very similar proof to that outlined above). The point is that the definition of $x + y$ given above recurses downwards on the value of x, building up the result in y, while $(x^+ + y) = (x^+ = 0 \rightarrow y, (x^+)^- + y^+)$ requires a test in the equation for f which can simultaneously be satisfied by $z = 0$ and $x^+ = 0$. This point seems to support Dijkstra's (1972, section 8) argument that programs should only be considered equivalent if they embody structurally similar processes as well as computing the same function. The ability to match terminal tests implies a degree of such similarity, and will lead us to formulate a heuristic of expanding innermost function applications first, since function nesting provides a form of sequencing.

The previous paragraph refers to a confusion which is as old as computer science. Equation (1) is intended to be a definition of the familiar addition function, given in McCarthy's (1963) conditional expression formalism. Mathematics currently defines a function to be a set of ordered pairs, and so there is nothing to choose between equation (1) and any other description of the same function, that is, set of ordered pairs. In particular, there is no mathematical difference between equations (1) and (3). The difference between equations (1) and (3) referred to in the preceding paragraph arises when McCarthy's formalism is given an algorithmic interpretation. Thus as well as defining the ordered pair $\langle\langle 3, 4\rangle, 7\rangle$, we also think of equation 1 as giving rise to the computational sequence $\langle 3, 4\rangle$, $\langle 2, 5\rangle$, $\langle 1, 6\rangle$, $\langle 0, 7\rangle$, finally yielding the answer 7. Similarly, if f and g are functions, their composition $g(f(x))$ is also a function, and mathematics does not prescribe that the computation of an instance of $g(f(x))$ should first consist of an application of f, and one of g, although this is the usual algorithmic interpretation of composition. To be precise, recursion induction is a technique for proving the equality of

*The author would like to thank the referee for his helpful comments.
†that is, $f(p \rightarrow a, b)$ is equal to $p \rightarrow f(a), f(b)$.

two functions which have different descriptions. In the remainder of this paper we shall be content to note the above remarks, and continue to refer to the process embodied by a function. Elsewhere the author, together with Richard Bornat, has explored the above issue more deeply as part of a continuing critique of mathematical semantics (see Brady 1977, chapter 8). Bornat and Brady (1977) also contains more difficult proofs than those given here, whose purpose is not to establish interesting new theorems but to illustrate the idea that recursion induction proofs can be developed quite rationally hand in hand with the discovery of the equation called for by the technique.

Since McCarthy (1963), other techniques for proving equivalence have been discovered, notably computational induction (Park, 1969; De Bakker and Scott, 1969) and structural induction (Burstall, 1969; McCarthy and Painter, 1967). Burstall (1969, page 41) points out that structural induction is a reformulated special case of recursion induction which seems, however, to be more easily applicable in the cases to which it applies. Boyer and Moore (1973) use structural induction in a program which automatically proves a large number of quite complex theorems about functions of *S*-expressions defined in McCarthy's formalism. We suggest that the greater clarity of structural induction, as well as the Boyer–Moore operations of reduce, normalise, and generalise, can be explained in terms of the analysis and matching described in this paper.

The author believes that the kinds of inferences described below could be automated; the use of a structural matcher and lemma generator suggest a close relationship between this study and Hardy's (1975) system for automatically programming LISP. Burstall and Darlington (1973, 1975) and Darlington (1975) discuss some related ideas aimed at optimising a recursive program or replacing a recursive program by an equivalent iterative one. The idea of automating the discovery of mathematical proofs is discussed by Moses (1967) and Bundy (1975).

The remainder of the paper consists of the detailed working of three examples in support of our claim that the equation for *f* can be discovered as the proof is discovered.

The theorems we shall consider involve the functions +, defined above, as well as length, append and reverse, which apply to single level lists, and which may be defined as follows:

$$\text{length}\,[l] = \text{null}\,[l] \to 0,\ \text{length}\,[\text{cdr}\,[l]]^+$$
$$\text{append}\,[l, m] = \text{null}\,[l] \to m,\ \text{cons}\,[\text{car}\,[l], \text{append}\,[\text{cdr}\,[l], m]]$$
$$\text{reverse}\,[l] = \text{null}\,[l] \to \text{NIL},\ \text{append}\,[\text{reverse}\,[\text{cdr}\,[l]], \text{list}\,[\text{car}\,[l]]]$$

where list $[x]$ = cons $[x, \text{NIL}]$. We shall also need the following properties of *S*-expressions as well as various properties of conditional expressions (see McCarthy, 1963, page 62).

$$\text{car}\,[\text{cons}\,[a, b]] = a$$
$$\text{cdr}\,[\text{cons}\,[a, b]] = b$$
$$\text{cons}\,[\text{car}\,[a], \text{cdr}\,[a]] = a, \text{ if } a \text{ is not atomic}$$
$$\text{null}\,[\text{cons}\,[a, b]] \text{ is false}.$$

Theorem 1

length [append $[l, m]$] = length $[l]$ + length $[m]$

We shall attempt to prove Theorem 1 by recursion induction: define $g\,[l, m]$ = length [append $[l, m]$] and $h\,[l, m]$ = length $[l]$ + length $[m]$. Clearly we are going to have to expand an instance of each of length, append and + at some stage of the proof, since the theorem would otherwise hold more generally; the question is when. There are two possible expansions on the left hand side, respectively yielding

$$g\,[l, m] = \text{null}\,[\text{append}\,[l, m]] \to 0,\ \text{length}\,[\text{cdr}\,[\text{append}\,[l, m]]]^+ \quad \text{(L1)}$$

$$g[l, m] = \text{null}\,[l] \to \text{length}\,[m],\ \text{length}\,[\text{cons}\,[\text{car}\,[l], \text{append}\,[\text{cdr}\,[l], m]]] \quad \text{(L2)}$$

There are three possibilities on the right hand side, two of which involve expanding instances of length.

$$h\,[l, m] = \text{length}\,[l] = 0 \to \text{length}\,[m],\ \text{length}\,[l]^- + \text{length}\,[m]^+ \quad \text{(R1)}$$
$$h\,[l, m] = \text{null}\,[l] \to \text{length}\,[m],\ \text{length}\,[\text{cdr}\,[l]]^+ + \text{length}\,[m] \quad \text{(R2)}$$
$$h\,[l, m] = \text{null}\,[m] \to \text{length}\,[l],\ \text{length}\,[l] + \text{length}\,[\text{cdr}\,[m]]^+ \quad \text{(R3)}$$

The best match is (L2) with (R2), since it immediately disposes of the terminal test and 'suggests' recursive calls with arguments cdr $[l]$ and m. That is, we attempt to find some function k which enables us to transform (L2) into

$$g\,[l, m] = \text{null}\,[l] \to \text{length}\,[m],\ k\,[g\,[\text{cdr}\,[l], m]]$$

and (R2) into

$$h\,[l, m] = \text{null}\,[l] \to \text{length}\,[m],\ k\,[h\,[\text{cdr}\,[l], m]]$$

If we can successfully find such a k, the proof will be complete, with

$$f\,[l, m] = \text{null}\,[l] \to \text{length}\,[m],\ k\,[f\,[\text{cdr}\,[l], m]]$$

the equation needed formally in the recursion induction proof technique. Now

$$k\,[g\,[\text{cdr}\,[l], m]] = k\,[\text{length}\,[\text{append}\,[\text{cdr}\,[l], m]]],$$

whereas the second half of L2 is

$$\text{length}\,[\text{cons}\,[\text{car}\,[l], \text{append}\,[\text{cdr}\,[l], m]]]\ . \quad \text{(L3)}$$

We have still not used the definition of length. Applying the definition of length and the properties of *S*-expressions noted above, (L3) reduces to

$$\text{length}\,[\text{append}\,[\text{cdr}\,[l], m]]^+$$

which suggests that $k\,[n] = n^+$. Similarly,

$$k\,[h\,[\text{cdr}\,[l], m]] = k\,[\text{length}\,[\text{cdr}\,[l]] + \text{length}\,[m]],$$

whereas we have in (R2)

$$\text{length}\,[\text{cdr}\,[l]]^+ + \text{length}\,[m]\ .$$

The choice of $k[n] = n^+$ is thus satisfactory if we can prove $(a + b)^+ = (a^+) + b$, which matches the second Theorem about + above. Hence the equation for *f* which enables a proof by recursion induction is

$$f\,[l, m] = \text{null}\,[l] \to \text{length}\,[m], f\,[\text{cdr}\,[l], m]^+\ .$$

Consider the expansions (L2) and (R2). There were two possible expansions on the left hand side; (L2) resulted from expanding the innermost—that is to say the *first* in terms of the interpretation of function nesting as sequencing. (R2) was chosen so that the test matched exactly, and more generally that the recursive call involved structurally similar function applications. These two observations amount to asserting that recursion induction proofs rely on being able to match program execution behaviour.

Theorem 2

reverse [append $[l, m]$] = append [reverse $[m]$, reverse $[l]$]

There is a choice of two function applications on the left hand side to expand; the 'sequence' heuristic used above suggests expanding the (inner) call on append to give

$$\text{null}\,[l] \to \text{reverse}\,[m],\ \text{reverse}\,[\text{cons}\,[\text{car}\,[l], \text{append}\,[\text{cdr}\,[l], m]]].$$

Expanding reverse, to move it past the call to cons so as to give a recursive call to $g[l, m]$ = reverse [append $[l, m]$], we get

$$\text{null}\,[l] \to \text{reverse}\,[m],\ \text{append}\,[\text{reverse}\,[\text{append}\,[\text{cdr}\,[l], m], \text{list}\,[\text{car}[l]]] =$$
$$\text{null}\,[l] \to \text{reverse}\,[m],$$

append [g [cdr [l], m], list [car [l]]] (L)

On the right hand side, we have the choice of expanding append, reverse [m] or reverse [l]. The terminal test 'null [l]', and the 'sequence' heuristic, suggest expanding reverse [l], which gives:

null [l] $\rightarrow$ append [reverse [m], NIL] ,
append [reverse [m], append [reverse [cdr [l]],
list [car [l]]]] (R)

matching the terminal test suggests a lemma:

append [L, NIL] = L ,

which is easily proved (by recursion or structural induction). There is no immediate instance of h in (R), although we might 'suspect' h [cdr [l], m], which ties in with (L). In fact, (L) strongly suggests the equation

$f[l, m]$ = null [l] $\rightarrow$ reverse [m],
append [f [cdr [l], m], list [car [l]]] (4)

which is satisfied by g. (4) would be satisfied by h if

append [h [cdr [l], m], list [car [l]]] (R2)

could be proved equal to

append [reverse [m], append [reverse [cdr [l]],
list [car [l]]]] .

Now (R2) is

append [append [reverse [m], reverse [cdr [l]]],
list [car [l]]] ,

which suggests the associativity of append as a lemma:

append [u, append [v, w]] = append [append [u, v], w]

This is easily shown using the techniques described in this paper.

Theorem 3
reverse [reverse [l]] = 1.
In this final case, $g[l]$ = reverse [reverse [l]] and $h[l]$ = 1, so that we have to work entirely from the left hand side to an equation for f. Using the 'sequence' heuristic to expand the innermost call to reverse, we get

reverse [reverse [l]] = null [l] $\rightarrow$ NIL,
reverse [append [reverse [cdr [l]],
list [car [l]]]] ,

since reverse [NIL] = NIL. Matching suggests we should work towards the recursive call reverse [reverse [cdr [l]]] of g, which involves interchanging reverse and append. The left hand side of Theorem 2 matches this subproblem, so we get

null [l] $\rightarrow$ NIL, append [reverse [list [car [l]]],
g [cdr [l]]] (L1)

which suggests that we should try to simplify

reverse [list [car [l]]] .

The sequence heuristic suggests expanding list (there is no obvious way to expand car), so we recall that list [m] = cons [m, NIL]. We can now expand reverse using the properties of car, cdr and cons given above, to get

append [reverse [NIL], list [car [l]]]

which is list [car [l]]. Substituting back in (L1) gives

null [l] $\rightarrow$ NIL, append [list [car [l]], g [cdr [l]]] (L2)

This can be tidied up by expanding list to get the equation

null [l] $\rightarrow$ NIL, cons [car [l], g [cdr [l]]] ,

which is clearly satisfied by h.

The discovery of an equation to serve in a proof by recursion induction does not have to rely on luck or inspiration, but can be rationally developed hand in hand with the development of the proof. A sequencing heuristic is suggested which reflects one facet of the idea that proofs of equivalence are only possible if the programs or functions to be proved equivalent have a reasonably similar process structure.

References

BORNAT, R. and BRADY, J. M. (1977). Recursion Induction considered harmful (submitted for publication).

BOYER, R. S. and MOORE, J. S. (1973). Proving theorems about LISP programs, *proceedings third Int. Jt. Conf. Art. Int.*, Stanford, pp. 486-493.

BRADY, J. M. (1977). *The theory of computer science:* a programming approach, Chapman and Hall.

BUNDY, A. (1975). Analysing mathematical proofs (or reading between the lines), *proc. fourth Int. Jt. Conf. Art. Int.*, Tblisı, pp. 22-28.

BURSTALL, R. M. (1969). Proving properties of programs by structural induction, *The Computer Journal*, Vol. 12, pp. 41-48.

BURSTALL, R. M. and DARLINGTON, J. (1973). A system which automatically improves programs, *Proc. third Int. Jt. Conf. Art. Int.* p. 479.

BURSTALL, R. M. and DARLINGTON, J. (1975). Some transformations for developing recursive programs, *Proc. Int. Conf. reliable software*, Los Angeles.

DARLINGTON, J. (1975). Application of program transformation to program synthesis, Colloquium on proving and improving programs, IRIA, France.

DE BAKKER, J. W. and SCOTT, D. (1969). A theory of programs, unpublished memo, Vienna.

DIJKSTRA, E. W. (1972). Notes on structured programming, in Dahl, Dijkstra and Hoare, *Structured programming*, Academic Press, pp. 1-82.

HARDY, S. (1975). Synthesis of LISP functions from examples, *proc. fourth Int. Jt. Conf. Art. Int.*, Tblisi, pp. 240-245.

KLEENE, S. C. (1950). *Introduction to metamathematics*, Van Nostrand.

MCCARTHY, J. (1963). *A basis for a mathematical science of computation, in Computer programming and formal systems*, (Braffert and Hirschberg eds.) North-Holland, pp. 33-70.

MCCARTHY, J. and PAINTER, J. A. (1967). Correctness of a compiler for arithmetical expressions, in *Math. aspects of computer science*, Amer. Math. Soc., RI, pp. 33-41.

MORRIS, J. H. Jr. (1971). Another recursion induction principle, *CACM*, Vol. 14, pp. 351-354.

MOSES, J. (1967). Symbolic Integration, Ph.D. thesis, MIT.

PARK, D. (1969). Fixpoint induction and proofs of program properties, *Machine Intelligence*, Vol. 5, (Meltzer and Mitchie eds.), Edinburgh University Press, pp. 59-78.