Proving properties of programs by structural induction

By R. M. Burstall*

This paper discusses the technique of structural induction for proving theorems about programs. This technique is closely related to recursion induction but makes use of the inductive definition of the data structures handled by the programs. It treats programs with recursion but without assignments or jumps. Some syntactic extensions to Landin's functional programming language ISWIM are suggested which make it easier to program the manipulation of data structures and to develop proofs about such programs. Two sample proofs are given to demonstrate the technique, one for a tree sorting algorithm and one for a simple compiler for expressions.

(First received April 1968 and in revised form August 1968)

Since the problem of proving that computer programs really do what their inventors allege them to do was discussed by McCarthy (1963), there has been considerable progress and proofs have been produced for non-trivial programs such as a simple compiler (Painter, 1967; Kaplan, 1967).

Three distinct methods of proof have been used.

(i) Recursion induction (McCarthy, 1963; Cooper, 1966; Kaplan, 1967).
(ii) Structural induction (McCarthy and Painter, 1967; Painter, 1967; Burstall, 1967).
(iii) Interpretation of flowcharts (Floyd, 1967).

The first two methods apply to programs where repetition is accomplished by recursion, and jumps and assignment are avoided. The third applies to programs using jumps and assignment but not recursion.

Recursion induction and structural induction are closely related. This paper discusses the latter. It contains the substance of a talk given at the conference on 'Mathematical Theory of Computation' at IBM Yorktown Heights. After that talk Professor McCarthy pointed out to me just how close is the parallel between these two methods, and produced an informal but convincing argument that any proof by structural induction can be turned into one by recursion induction. Thus in a sense structural induction is merely a special case of recursion induction, presented in a rather different manner.

I hope that the presentation of structural induction here will be of interest because

(i) structural induction is a very powerful tool,
(ii) it can be justified quite easily in terms of the ordering properties of the data structures denoted by the programs, using a well-known induction principle of algebra,
(iii) it is easy to use and seems to give a clear explanation in ordinary mathematical terms of why a program works as it does.

The proofs presented as examples will be mathematically rigorous but not formalised to the point where each inference is presented as a mechanical application of elementary rules of symbol manipulation. This is deliberate since I feel that our first aim should be to devise methods of proof which will prove the validity of non-trivial programs in a natural and intelligible manner. Obviously we will wish at some stage to formalise the reasoning to a point where it can be performed by a computer to give a mechanised debugging service. But we also need a method and notation which will make program proofs no more difficult to devise or read than other parts of mathematics, and which will give the same degree of insight into the reasoning involved. This is particularly important if we wish to communicate the hitherto intuitive reasoning processes of accomplished programmers to novices. The reasoning must be made explicit and rigorous but not necessarily mechanical. Although mechanised debugging is certainly very desirable, attempts to restrict the nature of the proofs devised to enable a computer to check them may delay the discovery of the variety of mathematical techniques which are applicable.

The main aim of this paper is to suggest some syntactic devices for writing programs in a way which makes it easier to derive proofs by structural induction. By cutting down some of the trivial but tedious manipulation involved the shape of the proof becomes easier to grasp. A pleasing property is that the form of the proofs is then very similar to that of the programs to which they refer. As a preliminary to this I have tried to clarify the principles on which structural induction rests and relate them to a well-known induction principle of algebra. The techniques are demonstrated by applying them to the proofs of correctness of a tree sorting algorithm and to the proof of correctness of a simple compiler for expressions.

If I have succeeded at all in simplifying the presentation and discovery of proof procedures I hope that a wider circle of programmers will become interested. I am convinced that the discipline of stating theorems and devising proofs will have a very beneficial effect on programming education and programming practice.

Structures and induction

The induction method which we will use in our proofs depends on the structure of the data objects which are manipulated by the program. We must prove not just that the program will work for specimen input data (the traditional method of debugging) but that it will work in general for *any* input data. To prove that it works for arbitrarily complex data it is natural to define the

* *Department of Machine Intelligence and Perception, University of Edinburgh*

data objects inductively. We then show that it works for the most elementary data, and that it will work for data of any degree of complexity provided that it works for all data of lesser complexity. We may then induce that it works for all data.

The programs will be written in a simple functional programming language; I have chosen to use and extend Landin's ISWIM (Landin, 1966). Before we discuss the language or notation, however, we must consider the kind of data object which will be denoted by expressions in the language, and specify the induction principles which may be used. In languages such as ALGOL 60 these objects are restricted to integers, reals, truth values and arrays of these. We will consider here some unspecified set of atomic objects and structures (often called 'records' or 'plexes') built out of these.

Thus any expression in the language will denote an *object* which will be either an *atom* or a *structure*. The atoms may be any set, finite or infinite. We will include functions amongst the atoms.

To define the structures we need a finite set of *construction operations* each of which takes a fixed number of objects as arguments. A structure is obtained by taking a sequence of objects (atoms or structures) and combining them with a construction operation. Thus each structure is built up from atoms by using a finite number of construction operations. For example if a_1, a_2, etc., are atoms, ω_1 is a 2-place construction operation and ω_2 a 1-place construction operation, then a_1, $\omega_1(a_1, a_1)$, $\omega_2(\omega_1(a_2, \omega_2(a_1)))$, etc., are structures.

We assume that there are some primitive functions given among the atoms and that we are able to define new ones in terms of them. Corresponding to each construction operation there are to be three functions.

(i) a constructor function, which given the components produces the structure as a result,
(ii) a destructor function, which given the structure produces the components as results, i.e. it produces an ordered n-tuple whose elements are the components,
(iii) a predicate function, which recognises whether an object has been made by using the given construction operation, producing a truth value as its result.

We assume an identity relation over the atoms and extend this to the structures as follows:

Two objects are identical if and only if they are identical atoms or they are obtained from identical components using the same construction operation.

This is a recursive definition of the extended identity in terms of the identity for atoms and the destructor and predicate functions. It may be convenient to define other equivalence operations but this is the basic identity. It implies that each structure has a unique set of components; in algebraic terms the structures are a word algebra.

As an example the natural numbers can be defined in our system by introducing an atom (or alternatively a construction operation with no arguments) called 0, and a construction operation with one argument called successor. We will then have functions to find the successor of a number (the constructor) to find its predecessor (the destructor) and to recognise a non-zero number (the predicate). In terms of those we define identity of numbers and then functions like addition and multiplication. Another example of structures would be lists in pure LISP (i.e. LISP without assignment). Circular lists (rings) would, however, be excluded.

What kind of induction principle can be employ in such a system?

We define a relation 'constituent' between objects as follows.

A is a *constituent* of B if A is identical with B or if A is a constituent of a component of B.

Thus the constituents of $\omega_2(\omega_1(a_2, \omega_2(a_1)))$ are a_1, a_2, $\omega_2(a_1)$, $\omega_1(a_2, \omega_2(a_1))$ and $\omega_2(\omega_1(a_2, \omega_2(a_1)))$. We may call a constituent of an object which is not identical to it a *proper constituent*, i.e. all except the last are proper constituents.

Now we may state an induction principle:

If for some set of structures a structure has a certain property whenever all its proper constituents have that property then all the structures in the set have the property.

In the special case of an atomic structure which has no proper constituents the clause 'all its proper constituents have that property' is trivially true and we just have to show that the atomic structure has the required property. For example, to prove that all lists have a certain property we show that a list has the property provided that all its sublists have the property, and in particular that the null list (which has no sublists) has the property. This is the induction principle used by McCarthy and Painter (1966). Curry and Feys (1958) call it 'structural induction'. It has been used widely by logicians, e.g. to prove the deduction theorem for propositional calculus by induction on the structure of formulas of that calculus.

We can make use of a rather more general principle to do a wider variety of inductions which has the above as a special case. The generalisation is not essential for the understanding of the notation and examples which follow, and some readers may prefer to omit it at first reading.

We first note that the constituent relation is a (partial) ordering. Now if $\leqslant$ is a (partial) ordering over a set A, a is said to be a *minimal element* of A if there is no a' in A such that $a' < a$. We say that A satisfies the *minimum condition* if every non-empty subset has a minimal element. I have taken these definitions from Cohn (1965) who states the following principle:

'*Generalised principle of induction* (*Noetherian induction*)

Let A be an ordered set with minimum condition and B a subset of A which contains any element $a \in A$ whenever it contains all the elements $x \in A$ such that $x < a$. Then $B = A$.'

This is the principle which we need to prove theorems about structures using the relation 'constituent', which is an ordering which ensures that any set of objects satisfies the minimum condition. This follows since each structure has a finite number of components and is built up by a finite number of construction operations, and no structure is identical to a constituent of itself.

We can apply the generalised induction principle to our objects using the constituent relation itself or indeed using any other relation defined in terms of it which satisfies the minimum condition. For example, it may

also be convenient to do induction on n-tuples of objects by defining an ordering on n-tuples in terms of the constituent relation.

The generalised induction principle differs in two respects from the usual 'course of values' induction: it allows for partial ordering, not just total ordering, and it allows induction over the transfinite steps. Only the first of these will be used here.

Before passing on to the question of a convenient language for programs about structures which will be easy to validate programs, it is worth making a remark about assignment. Assignment is a feature of programming languages which is not reflected in our discussion. Data structures in a system with assignment are quite different from the structures described above; for example, they may share components or even be circular, and there are two interesting equivalences on them (identity of address and isomorphism). They can be represented in our system by a method which I have discussed elsewhere (Burstall, 1968). One hopes also that it may be possible to combine Floyd's method of obtaining proofs for programs with assignment with the methods of structural induction or recursion induction.

Some language devices for manipulating structures

In what follows I will take the simple functional language ISWIM (Landin, 1966) as a basis and suggest some syntactic extensions for the convenient manipulation of the structures described in the last section. A principal aim of these extensions is to make it easier to prove theorems about the programs. In fact the notation so arranges the programs that the proof can mirror the program layout very closely. It also gets rid of extraneous names and the necessity of appealing to a set of axioms which are used purely to specify the interrelationship of the names.

ISWIM is based on LISP but is more regular and in some ways more convenient. It may best be regarded as a palatable syntactic dress for Church's lambda calculus, and the conversion rules carry over from that calculus. This close relationship to a calculus with known logical properties makes it suitable for us here. I will not give a description of ISWIM or its conversion rules, referring the reader to Landin's paper. In another paper (Burstall, 1968) I give an informal description of ISWIM with some examples. Here I am concerned to extend the language and the conversion rules to deal more conveniently with data structures.

An example will serve to introduce the ISWIM notation, a function to concatenate lists, e.g. *concat* ((1, 2, 3), (4, 5)) = (1, 2, 3, 4, 5)

let rec *concat*($xs1$, $xs2$) = **if** *null* ($xs1$) **then** $xs2$
else let x = *car*($xs1$) **and** $xs3$ = *cdr*($xs1$)
cons(x, *concat*($xs3$, $xs2$))

Here **let** introduces the definition of a variable or a function. Simultaneous definition is accomplished by **let** . . . **and** . . . . A recursive definition is indicated by **rec**.

ISWIM also allows a sequence of identifiers on the left of a definition provided that the expression on the right produces an n-tuple of values rather than just one, e.g.

let (x, y) = *splitup*(l)

It even allows nested expressions such as (x,(y, z)) on the left of a definition, but I will not use this facility, precisely because I wish to generalise it to handle arbitrary data structures as well as simple n-tuples.

In proving theorems we wish to convert expressions in an ISWIM program into other equivalent expressions, e.g.

$$\left.\begin{array}{l}\textbf{let } x = u + 3\\ x^2 + z\end{array}\right\} \quad \text{into} \quad \left\{\begin{array}{l}\textbf{let } y = u + 3\\ y^2 + z\end{array}\right.$$

or

$$\left.\begin{array}{l}\textbf{let } x = u + 3\\ x^2 + z\end{array}\right\} \quad \text{into} \quad \left\{ (u + 3)^2 + z\right.$$

In general we have

$$\left.\begin{array}{l}(\text{Rule } \alpha)\ \textbf{let } x = E1\\ \qquad E2\end{array}\right\} \text{converts to} \left\{\begin{array}{l}\textbf{let } y = E1\\ \text{substitute } y \text{ for } x \text{ in } E2\end{array}\right.$$

$$\left.\begin{array}{l}(\text{Rule } \beta)\ \textbf{let } \omega = \text{E1}\\ \qquad E2\end{array}\right\} \text{converts to} \left\{ \text{substitute } E1 \text{ for } x \text{ in } E2\right.$$

(A technicality: we demand that the substitution does not cause y (in rule α) or a variable of $E1$ (in rule β) to become bound in $E2$.)

The rules for the manipulation of conditionals are almost self-evident; for formal details see McCarthy (1963), e.g. given $E1$ = *true* we have

if $E1$ **then** $E2$ **else** $E3$ } converts to { $E2$.

Let us now turn to the extensions for conveniently manipulating data structures. Consider, for example, the treatment of lists in LISP and languages derived from LISP. There is an object

nil

and 5 functions

cons
car, *cdr*
atom
null

We might use instead of cons, car, cdr and atom, the 3 functions

cons	the constructor, to make a list
decons	the destructor, to produce the *car* and the *cdr* of the list
compound	the predicate, *not atom*.

I propose further that a single identifier, say *cons*, should serve for them all, which one of the three, or whether all three at once are meant being determined by context. Thus

Extended ISWIM		ISWIM
let x = *cons* (a, y)	means	**let** x = *cons*(a, y)
let *cons* (a, y) = x	means	**let** (a, y) = *decons*(x)
x **is** *cons*	means	*compound*(x)
f(*cons*)	means	*f*(*cons*, *decons*, *compound*)

In the last example *cons* denotes a single entity which comprises all three functions, i.e. a special kind of 3-component structure. This is what is passed on as an actual parameter.

For example, the concatenation of lists (the function

concat defined above) would be defined in our extended ISWIM

let rec *concat*(*xs*1, *xs*2) = **if** *xs*1 **is** *cons*
then let *cons*(*x*, *xs*3) = *xs*1
cons (*x*, *concat*(*xs*3,*xs*2))
else *xs*2

Similarly we may treat the empty list as a structure with no components and denote all three functions involved by *nil*. Thus

let *x* = *nil*( )
let *nil*( ) = *x* (a declaration of no variables!)
x **is** *nil*
f(*nil*)

Turning to theorem proving we now see that instead of the 4 axioms

(i) *cons*(*car*(*x*), *cdr*(*x*)) = *x*
(ii) *car*(*cons*(*x*, *y*)) = *x*
(iii) *cdr*(*cons*(*x*, *y*)) = *y*
(iv) *atom*(*cons*(*x*, *y*)) = *false*

we have the substitution rules (using → for 'converts into')

(i) **let** *cons*(x_1, x_2) = *cons*(e_1, e_2) → $\phi[e_1, e_2]$
$\phi[x_1, x_2]$
(ii) *cons*(e_1, e_2) **is** *cons* → *true*

Here the x_i are variables, the e_i are any expressions and $\phi[\alpha_i, \ldots, \alpha_k]$ is any expression involving $\alpha_1, \ldots, \alpha_k$.

What is important is that these rules can be stated quite generally for any construction operation *c*. The set of substitution rules needed for dealing with constructs is:

(i) **let** $c(x_1, \ldots, x_k) = c(e_1 \ldots, e_k)$; $\phi[x_1, \ldots, x_k]$ → $\phi[e_1, \ldots, e_k]$
(ii) $c'(e_1, \ldots, e_k)$ **is** *c* → *true if* $c' = c$ else *false*
(iii) *a* **is** *c* → *false if a* is an atom.

Thus we can apply simplifying transformations to an extended ISWIM expression by looking at its structure without reference to any external axioms, provided only that we know which identifiers correspond to construction operations.

There is no reason why we should not allow more complex left-hand sides of definitions, e.g.

let *cons*(*x*1, *cons*(*x*2, *xs*2)) = *xs*1

Rule (*i*) must be extended accordingly

(i) **let** $\psi(x_1, \ldots, x_k, c_1, \ldots, c_n) = \psi(e_1, \ldots, e_k, c_1 \ldots, c_n)$ $\phi[x_1, \ldots, x_k]$ → $\phi[e_1, \ldots, e_n]$

where ψ is any expression formed entirely from the $x_1, \ldots, x_k$ and the construction operations $c_1, \ldots, c_n$.

This idea of complex left-hand sides for definitions is an extension of the ISWIM device for *n*-tuples (lists), but it is much more powerful if allowed for any structures. Brooker (1966) has an analogous device for resolving structures into components, based on his compiler compiler notation.

A further extension gives added simplicity. R. J. Popplestone (private communication) pointed out to me that in a case such as the definition of *concat* above there is some redundancy in writing

if *xs*1 **is** *cons* **then let** *cons*(*x*, *xs*3) = *xs*1; . . .

and it could well be replaced by some form mentioning *cons* and *xs*1 only once.

I propose that we abbreviate

if *e* **is** c_1 **then let** $c_1(x_1, \ldots, x_{k_1}) = e$; $\phi_1(x_1, \ldots, x_{k_1})$
else if *e* **is** c_2 **then let** $c_2(x_1, \ldots, x_{k_2}) = e$; $\phi_2(x_1, \ldots, x_{k_2})$
. . . . . .
else if *e* **is** c_n **then let** $c_n(x_1, \ldots, x_{k_n}) = e$; $\phi_n(x_1, \ldots, x_{k_n})$

to

cases *e*:
$c_1(x_1, \ldots, x_{k_1})$: $\phi_1(x_1, \ldots, x_{k_1})$
$c_2(x_1, \ldots, x_{k_2})$: $\phi_2(x_1, \ldots, x_{k_2})$
. . . . .
$c_n(x_1, \ldots, x_{k_n})$: $\phi_n(x_1, \ldots, x_{k_n})$

The definition of *concat* now reads

let rec *concat*(*xs*1, *xs*2) = **cases** *xs*1:
cons (*x*, *xs*3): *cons* (*x*, *concat* (*xs*3, *xs*2))
nil( ) : *xs*2

These **cases** expressions were partly suggested by a **case** switch on type introduced into CPL by M. Richards (1967).

The conversion rules stated above carry over to **cases** expressions. Thus in the above **cases** expression if *e* is $c_i(e_1, \ldots, e_{k_i})$ where c_i is in the set $c_1, \ldots, c_n$ then the whole expression converts to $\phi_i(e_1, \ldots, e_{k_i})$.

A small point remains to be specified before giving an example of a proof using these techniques: the means of introducing construction operations into a program. I suggest that each construction operation introduce a new type and that disjunctions of these types should also be types, e.g. (roughly following Landin 1964)

a *cons* has an *atom* and a *list*
a *nil* has no components
a *list* is a *cons* or a *nil*.

Two lemmas

In the next two sections I will prove a simple theorem about compiling expressions, and a theorem about a sorting program. But before doing so it will be convenient first to define two list processing functions and prove two lemmas about them. We will consider lists of αs where α is any type of object.

An α-*list* is either a *cons* or a *nil*.
A *cons* has an α and a *list*.
A *nil* has no components.

We will use an infixed :: instead of *cons*, writing $x :: xs$ for *cons*(*x*, *xs*). (This is not to be confused with the : used in the layout of **cases** expressions.)

First we repeat the definition of *concat* and then define *lit*, a function to apply a two-argument function to all elements of a list with a given starting value (Barron and Strachey, 1966).

let rec *concat*(*xs*1, *xs*2) = **cases** *xs*1:
x :: *xs*1: *x* :: *concat*(*xs*1, *xs*2)
nil( ) : *xs*2

Note that *xs*1 has been used in the second line instead of the *xs*3 used previously. This causes no confusion with the parameter *xs*1 since this new *xs*1 is a local variable with scope confined to the expression *x* :: *concat* (*xs*1, *xs*2). Such puns emphasise the structure of the recursion and I shall use them freely.

let rec *lit*(*f*, *xs*, *y*) = **cases** *xs*:
x :: *xs*: *f*(*x*, *lit*(*f*, *xs*, *y*))
nil( ): *y*

Example *lit*(+, (2, 3, 4), 1) = (2+ (3+ (4 + 1))) = 10

Lemma *lit* (*f*, *concat*(*xs*1, *xs*2), *y*) = *lit*(*f*, *xs*1, *lit*(*f*, *xs*2, *y*))
Proof by structural induction on lists
cases *xs*1:
x :: *xs*1:

LHS = *lit* (*f*, *x* :: *concat* (*xs*1, *xs*2), *y*)	by defn. of *concat*
= *f*(*x*, *lit*(*f*, *concat* (*xs*1, *xs*2) ,*y*))	by defn. of *lit*
RHS = *f*(*x*, *lit* (*f*, *xs*1, *lit* (*f*, *xs*2, *y*)))	by defn. of *lit*
= *f*(*x*, *lit*(*f*, *concat* (*xs*1, *xs*2) ,*y*))	by Induction Hypothesis

nil( ):

LHS = *lit*(*f*, *xs*2, *y*)	by defn. of *concat*
RHS = *lit* (*f*, *xs*2, *y*)	by defn. of *lit*

Lemma
Suppose *xs* is an *α-list* and *P* a property.
If $P(y_0)$, and $P(y)$ implies $P(f(x, y))$ for any $x \in \alpha$
then $P(lit(f, xs, y_0))$

Proof $P(lit(f, xs, y_0)) =$
cases *xs*:
nil( ) : $P(y_0) = true$
x :: *xs* : $f(x, lit(f, xs, y_0)) = true$ by induction hyp.

These proofs have a general layout very similar to a function which deals with lists. I have not given any formal rules for this layout, nor do I wish to do so. We see, however, that as well as the display of a sequence of cases the use of local variables in the proof can be very convenient. Thus when we say *x* :: *xs*1, as in the first case, we mean 'suppose that *x* is of the form *x* :: *xs*1' here the new *xs*1 is a local variable to that case, and it is convenient to re-use the name *xs*1 for it. Although it should be possible to formalise the rules for this style of proof I think that a little more informal exploration should be done first.

Proof of a tree sort program

As an example of the use of these techniques I will prove the correctness of a program which sorts a list by converting it into an ordered tree and then back to a list. First we need data definitions.

Data definitions

A *list* is a *cons*
or *nil*
A *cons* has an *item*
and a *list*
nil has no components
An *item* is atomic
A *tree* is a *node*
or a *tip*
or *niltree*
A *node* has a *tree*
and an *item*
and a *tree*
A *tip* has an *item*
niltree has no components.

The items have an ordering $\leqslant$ defined over them. The item held at a node of the tree is chosen so that the left subtree has items not greater than it and the right subtree has items not less than it.

Program

The function *sort* sorts a list of items into the order defined by $\leqslant$. It does so by constructing an ordered tree and then flattening it into a list. A node of the tree contains two subtrees and an item. All the items in the left subtree are $\leqslant$ the items and the item is $\leqslant$ all items in the right subtree. An example is shown in **Fig. 1.**

The function *totree* adds an item to a tree.

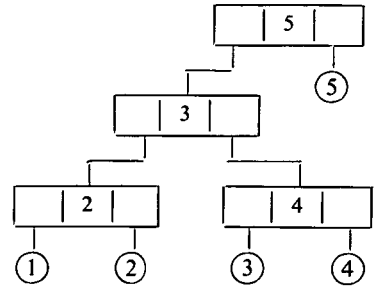

Fig. 1. Tree obtained in sorting the list (3, 5, 2, 1, 4) into the list (1, 2, 3, 4, 5)

let rec *totree*(*i*, *t*) =
cases *t*:
niltree( ): *tip*(*i*)
tip(*i*1) : **if** *i*1 $\leqslant$ *i*
then *node* (*t*, *i*, *tip*(*i*))
else *node*(*tip*(*i*), *i*1, *t*)
node(*t*1, *i*1, *t*2): **if** *i*1 $\leqslant$ *i*
then *node* (*t*1, *i*1, *totree* (*i*, *t*2))
else *node*(*totree*(*i*, *t*1), *i*1, *t*2)

The function *maketree* converts a list to a tree.
let *maketree* (*is*) = *lit*(*totree*, *is*, *niltree*())

The function *flatten* converts a tree to a list.
let rec *flatten*(*t*) =
cases *t*:
niltree( ): *nil*( )
tip (*i*) : *i* :: *nil*( )
node(*t*1, *i*1, *t*2): *concat*(*flatten*(*t*1), *flatten*(*t*2))
let *sort*(*is*) = *flatten*(*maketree*(*is*))

Definitions

Before stating formally the theorem about *sort* we need some definitions. We define analogues of $\leqslant$ for lists and trees and defined the property of being ordered for lists and trees. We shall not scruple to use the same identifier for a family of analogous predicates, e.g. $\leqslant$ or *ord*, relying on the type of the arguments to make clear the meaning. The statement of the theorem only requires the definitions for lists; we give those for trees as well for use later in the proof.

let rec $i \leqslant is$ = **cases** *is*:
nil( ) : *true*
$i1$ :: *is* : $i \leqslant i1$ *and* $i \leqslant is$

let rec $is1 \leqslant is2$ = **cases** $is1$:
nil( ) : *true*
$i1$ :: $is1$: $i1 \leqslant is2$ **and** $is1 \leqslant is2$

let rec *ord*(*is*) = **cases** *is* :
nil( ) : *true*
i :: *is* : $i \leqslant is$ **and** *ord*(*is*)

let rec $i \leqslant t$ = **cases** *t* :
niltree( ) : *true*
tip($i1$) : $i \leqslant i1$
node($t1$, $i1$, $t2$) : $i \leqslant t1$ **and** $i \leqslant t2$

let rec $t1 \leqslant t2$ = **cases** $t1$:
niltree( ) : *true*
tip(*i*) : $i \leqslant t2$
node($t11$, *i*, $t12$) : $t11 \leqslant t2$ **and**
$t2 \leqslant t12$

let rec *ord*($t1$) = **cases** *t* :
niltree( ) : *true*
tip(*i*) : *true*
node ($t1$, *i*, $t2$) : *ord*($t1$) **and** *ord*($t2$)
and $t1 \leqslant i$ **and** $i \leqslant t2$

(Note. The relation $\leqslant$ between lists is only an ordering if *nil* is excluded, similarly for trees. This is untidy but will not cause trouble.)

Theorem *ord*(*sort*(*is*)). This states that *sort* produces an ordered list.

Proof
Lemma If *t* is ordered then *totree*(*i*, *t*) is ordered
Proof *ord*(*totree*(*i*, *t*)) =
cases *t*:
niltree( ) : *ord*(*tip*(*i*)) by def. of *totree*
= **true** by def. of *ord*
tip($i1$) : *ord*(**if** $i1 \leqslant i$ **then** *node*(*t*, *i*, *tip*(*i*))
else *node*(*tip*(*i*), $i1$, *t*))
by def. of *totree*
= *true* in either case
by def. of *ord*
node($t1$, $i1$, $t2$): *ord*(**if** $i1 \leqslant i$ **then** *node* ($t1$, $i1$,
totree(*i*, $t2$))
else *node*(*totree*(*i*, $t1$)
$i1$, $t2$)
by def. of *totree*
= *true* in either case, since *t* is ordered
and using def. of *ord*.

Lemma *ord*(*maketree*(*is*))
Proof *maketree*(*is*) = *lit*(*totree*, *is*, *niltree*( )),
and *ord* (*niltree*( ))
we use the lemma about *totree* and the lemma about *lit* (in the last section)
Lemma if *ord*($is1$) and *ord*($is2$) and $is1 \leqslant is2$ then *ord*(*concat*($is1$, $is2$))
Proof *ord*(*concat*($is1$, $is2$)) = **cases** $is1$:
nil( ): *ord*($is2$) by def. of *concat*
= *true*
$i1$ :: $is1$: *ord*($i1$ :: *concat*($is1$, $is2$)) by def. of *concat*
= *true* since *ord*(*concat*($is1$, $is2$)) by ind. hyp.
and by hypothesis $i1 \leqslant$ is1 and $is1 \leqslant is2$.
Lemma if *ord*(*t*) then *ord*(*flatten*(*t*))
Proof *ord*(*flatten*(*t*)) = **cases** *t*:
niltree( ): *ord*(*nil*( )) = *true*
tip(*i*) : *ord*(*i* :: *nil*( )) = *true*
node($t1$, *i*, $t2$) = *ord*(*concat*(*flatten*($t1$), *flatten*($t2$))
= *true* by lemma on *concat* and ind. hyp.

We now prove the theorem immediately since *ord*(*maketree*(*is*)) and hence *ord*(*flatten*(*maketree*(*is*))), i.e. *ord*(*sort*(*is*)).

Proof of correctness of a compiler for expressions

To demonstrate the use of the devices which we have suggested for manipulating structures I will now prove that expressions can be evaluated by executing a sequence of instructions which act upon a stack. This will be a simple exercise in the same vein as the proof of the correctness of a compiler for expressions by McCarthy and Painter (1966).

The atoms and constructs which we require (in addition to lists as previously defined) will be as follows:

Source language

An *expression* is a *compound* or an *identexpr* or a *constexpr*.
A *compound* has an *operator* and an *expression* and an *expression*.
An *identexpr* has an *identifier*.
A *constexpr* has a *constant*.
Constants, *identifiers* and *operators* are *atoms*.

Semantics

The expressions will denote 'items', as will the constants. The operators will denote functions of items (we consider only 2-argument functions for simplicity). The nature of the items and the particular functions need not be specified. The identifiers will denote 'variables' and we are given a function which associates each variable with an item (this is the so-called 'state vector')

items are *atoms*

There are the following functions (we use $x_1, \ldots x_n \Rightarrow y$ to denote the set of all functions from $x_1 \times \ldots \times x_n$ to y):

itemof ϵ *const* $\Rightarrow$ *item*
varof ϵ *identifier* $\Rightarrow$ *variable*
funcof ϵ *operator* $\Rightarrow$ (*item*, *item* $\Rightarrow$ *item*)
varvalue ϵ *variable* $\Rightarrow$ *item* (Note. This is the 'state vector' or 'environment'.)

Machine language

There are three instructions: to perform a binary operation on the top of the stack, to load the value of a

variable onto the stack, to load (literally) the value of a constant onto the stack.

An *instruction* is an *operate* or a *loadident* or a *loadconst*.
An *operate* has an *operation*.
A *loadident* has an *identifier*.
A *loadconst* has a *constant*.
An *mprogram* is a *list* of *instructions*.

Semantics

A *stack* is a *list* of *items*.

We first define the value of a source language expression, then the value of a machine program in terms of a function to execute an individual instruction and produce a new stack. Thus

val ∈ *expression* ⇒ *item*
do ∈ *instruction*, *stack* ⇒ *stack*
mpval ∈ *mprogram* ⇒ (*stack* ⇒ *stack*)

We assume a fixed function *varval* throughout and do not include it as a parameter.

```
let rec val (e) = cases e:
                    compound(op, e1, e2) :
                        funcof(op) (val(e1),val(e2))
                    identexpr(id)        : varvalue
                                           (varof(id))
                    constexpr(c)         : itemof(c)
let do(in, st)  = cases in:
                    operate(op): let i1 :: i2 :: st = st;
                                     funcof(op)(i1, i2) :: st
                    loadident(id) : varvalue(varof(id)) :: st
                    loadconst(c)  : itemof(c)           :: st
let mpval(mp) = λst.lit(do, mp, st)
```

We compile an expression into a list of instructions using

```
comp ∈ expression ⇒ mprogram
let rec comp(e) = cases e:
        compound(op, e1, e2):
          operate(op) :: concat(comp(e1), comp(e2))
          identexpr(id) : loadident(id) :: nil( )
          constexpr(c)  : loadconst(c)  :: nil( )
```

Theorem The compilation establishes the correct correspondence between *val* and *mpval*, i.e.

$$mpval(comp(e))(s) = val(e) :: s$$

Thus running a compiled expression on the machine causes its value to be loaded onto the stack (see **Fig. 2**).

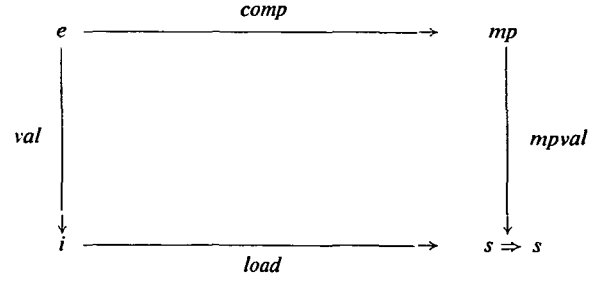

Fig. 2. Correctness of compilation. The condition is that this diagram commutes. The function *load* is *load* (*i*) = λ*s.i* :: *s*

Although the theorems are rather easy ones I hope that they will suffice to show the extra perspicuity obtained by using only one identifier to refer to each construction operation and by the **cases** notation as an extension to ISWIM.

Proof by structural induction on *e*

cases *e*:

compound(*op*, *e*1, *e*2):

LHS = *lit*(*do*, *comp*(*e*), *s*)	
= *lit*(*do*, *operate*(*op*) :: *concat*(*comp*(*e*1), *comp*(*e*2)), *s*)	by defn. of *mpval*
= *do*(*operate*(*op*), *lit*(*do*, *concat*(*comp*(*e*1), *comp*(*e*2)), *s*))	by defn. of *comp*
= *do*(*operate*(*op*), *lit*(*do*, *comp*(*e*1), *lit*(*do*, *comp*(*e*2), *s*)))	by defn. of *lit*
= *do*(*operate*(*op*), *mpval*(*e*1)(*mpval*(*e*2)(*s*)))	by the lemma in 'Two lemmas'
= *do*(*operate*(*op*), *val*(*e*1) :: *val*(*e*2) :: *s*)	by defn. of *mpval*
= *funcof*(*op*)(*val*(*e*1), *val*(*e*2)) :: *s*	by Induction Hyp.
RHS = *funcof*(*op*)(*val*(*e*1), *val*(*e*2)) :: *s*	by def. of *do*
	by defn. of *val*

identexpr(*id*):

LHS = *lit*(*do*, *comp*(*e*), *s*)	by defn. of *mpval*
= *lit*(*do*, *loadident*(*id*) :: *nil*( ), *s*)	by defn. of *comp*
= *do*(*loadident*(*id*), *s*)	by defn. of *lit*, twice
= *varvalue*(*varof*(*id*)) :: *s*	by defn. of *do*
RHS = *varvalue*(*varof*(*id*)) :: *s*	by defn. of *val*

constexpr(*c*):

LHS = *lit*(*do*, *comp*(*e*), *s*)	by defn. of *mpval*
= *lit*(*do*, *loadconst*(*c*) :: *nil*( ), *s*)	by defn. of *comp*
= *do*(*loadconst*(*c*), *s*)	by defn. of *lit*, twice
= *itemof*(*c*) :: *s*	by defn. of *do*
RHS = *itemof*(*c*) :: *s*	by defn. of *val*

This completes the proof.

Acknowledgements

I am grateful to P. J. Landin and C. Strachey for many illuminating discussions about the theory of programming, and to Professor J. A. Robinson for helping me to clarify the ideas presented here. The ideas are based on the work of Professor J. McCarthy. I would also like to thank IBM Corporation for making it possible for me to attend the conference on the Mathematical Theory of Computation and give the talk from which this paper is derived.

References

BARRON, D. W., and STRACHEY, C. (1966). Programming, *Advances in Programming and Non-numerical Computation* (ed. L. Fox) pp. 49–82.

BROOKER, R. A., and ROHL, J. S. (1967). Simply partitioned data structures: the compiler-compiler re-examined, *Machine Intelligence* 1 (eds. N. L. Collins and D. Michie). Edinburgh: Oliver and Boyd, pp. 229–239.

BURSTALL, R. M. (1968). Semantics of assignment, *Machine Intelligence* 2 (eds. E. Dale and D. Michie). Edinburgh: Oliver and Boyd, pp. 3–20.

COOPER, D. C. (1966). The equivalence of certain computations, *Computer Journal*, Vol. 9, pp. 45–52.

COHN, P. M. (1965). *Universal algebra*, New York and London: Harper and Row.

CURRY, H. B., and FEYS, R. (1958). *Combinatory logic*, Amsterdam: North Holland.

FLOYD, R. W. (1967). Assigning meanings to program, *Mathematical Aspects of Computer Science*. Amer. Math. Soc. Providence, Rhode Island, pp. 19–32.

KAPLAN, D. M. (1967). Correctness of a compiler for Algol-like programs, *Stanford Artificial Intelligence Memo. No.* 48, Department of Computer Science, Stanford University.

LANDIN, P. J. (1964). The mechanical evaluation of expressions, *Computer Journal*, Vol. 6, pp. 308–320.

LANDIN, P. J. (1966). The next 700 programming languages, *Comm. Ass. Comp. Mach.*, Vol. 9, pp. 157–166.

MCCARTHY, J. (1963). A basis for a mathematical theory of computation, *Computer Programming and Formal Systems* (eds. Braffort and Hirschberg), Amsterdam: North Holland, pp. 33–70.

MCCARTHY, J., and PAINTER, J. A. (1967). Correctness of a compiler for arithmetic expressions, *Mathematical Aspects of Computer Science*. Amer. Math. Soc. Providence, Rhode Island, pp. 33–41.

PAINTER, J. A. (1967). Semantic correctness of a compiler for an Algol-like language, *Stanford Artificial Intelligence Memo. No.* 44 (March 1967), Department of Computer Science, Stanford University.

RICHARDS, M. (1967), Basic CPL reference manual, Memo. M-352, Project MAC, M.I.T.

Note added in proof. Some further work on the topic of this paper is reported in:

LANDIN, P. J., and BURSTALL, R. M. (1969). Programs and their proofs: an algebraic approach. To appear in *Machine Intelligence 4* (eds. B. Meltzer and D. Michie). Edinburgh: University Press.

A program for solving word sum puzzles

By R. M. Burstall*

This paper describes a program for solving a class of 'word sum' or 'cryptarithm' puzzles by a heuristic tree searching method. Formally the problem is to solve a set of simultaneous linear inequalities with the variables taking integer values.

(First received October 1967 and in revised form May 1968)

1. Introduction

This paper describes a heuristic program for solving a class of 'word sum' problems sometimes called 'cryptarithms' (Brookes, 1964), which have attracted some attention as a simple example of problems which can be solved by brute enumeration but which can better be tackled with heuristic search reducing strategies. An example (with acknowledgements to English Electric) is

$$\begin{array}{r} K\ D\ F\ 9 \\ K\ D\ N\ 2 \\ K\ D\ F\ 6 \\ K\ D\ P\ 1\ 0 \\ \hline C\ A\ R\ E\ E\ R \end{array}$$

Each letter is to be replaced by a different digit to form a correct addition sum.

Enumeration means scanning 10! possibilities. Can the computer adopt a more humane method?

2. Mathematical formulation

We can express the problem thus (adding variables for carries)

$$\begin{aligned} 9 + 2 + 6 &= R + 10v \\ v + 2F + N + 1 &= E + 10w \\ w + 3D + P &= E + 10y \\ y + 3K + D &= R + 10z \\ z + K &= A + 10C \end{aligned}$$

* *Department of Machine Intelligence and Perception, University of Edinburgh*