INDUCTION PRINCIPLES FOR CONTEXT-FREE LANGUAGES

Ward Douglas Maurer

INTRODUCTION

In proving assertions about programs, it is necessary to use induction principles in one form or another. Recursion induction was introduced by McCarthy [4] for the purpose of proving assertions about functions, in particular, functions defined by recursive conditional expressions. Structural induction was introduced by Burstall [1] for proving assertions about structures, such as lists and trees. The induction principles which we shall introduce here are useful for proving assertions about context-free languages and their semantic attributes, as this term is used by Knuth [3]. We shall introduce these principles informally and use them to prove the equivalence of two syntactic definitions of an unsigned integer, the equivalence of Knuth's two examples of synthesized and inherited attributes, and the validity of Hoare's first axiom (also known as the "back substitution rule") for a large class of programming languages.

A SIMPLE EXAMPLE

Consider the BNF rules ⟨alpha⟩ ::= ⟨digit⟩ | ⟨alpha⟩ ⟨digit⟩ and ⟨beta⟩ ::= ⟨digit⟩ | ⟨digit⟩ ⟨beta⟩. We wish to prove that ⟨alpha⟩ and ⟨beta⟩ are the same -- that is, that every alpha is a beta, and vice versa. Intuitively this is clear, because an alpha is defined here as a non-empty sequence of digits, and so is a beta, although in a different way. Our proof of this trivial fact will serve to give a simple example of the method of proof; more complex examples follow.

As a lemma, we first prove that ⟨beta⟩ ⟨digit⟩ $\subset$ ⟨beta⟩ -- that is, "Every beta followed by a digit is again a beta." The symbol $\subset$ denotes inclusion of sets, as usual. The set of all digits is denoted by ⟨digit⟩ and the set of all betas by ⟨beta⟩. Concatenation is denoted by a blank, and this operator extends to sets in the usual way; that is, X Y is the set of all x y , for all x, y in the sets X, Y.

The rule defining ⟨beta⟩ may clearly be rewritten, using this notation, as ⟨beta⟩ = ⟨digit⟩ $\cup$ ⟨digit⟩ ⟨beta⟩, where = denotes set equality and where concatenation binds more strongly than union. Thus the statement of the lemma is equivalent to the two statements ⟨digit⟩ ⟨digit⟩ $\subset$ ⟨beta⟩ and ⟨digit⟩ ⟨beta⟩ ⟨digit⟩ $\subset$ ⟨beta⟩. The first of these is obvious -- a digit followed by a digit is always a beta. (Formally, we might write ⟨digit⟩ ⟨digit⟩ $\subset$ ⟨digit⟩ ⟨beta⟩ $\subset$ ⟨beta⟩, where the two steps

are justified respectively by the two alternatives in the definition of ⟨beta⟩.) It is in the second of these statements that we use our induction principle. Note that the left side is a special case of ⟨beta⟩ ⟨digit⟩, the left side of the statement of the lemma -- but it also contains a "smaller" ⟨beta⟩ ⟨digit⟩ (which is preceded by a digit). By induction, then, the conclusion -- that is, the statement of the lemma -- may be assumed to hold for this smaller ⟨beta⟩ ⟨digit⟩, and the result is ⟨digit⟩ ⟨beta⟩ ⟨digit⟩ $\subset$ ⟨digit⟩ ⟨beta⟩ (where we use the obvious fact that the associativity of concatenation extends naturally to set-concatenation). Since we already know that ⟨digit⟩ ⟨beta⟩ $\subset$ ⟨beta⟩ by the definition of ⟨beta⟩, the lemma follows.

We are now ready to prove that ⟨alpha⟩ $\subset$ ⟨beta⟩ -- that is, "Every alpha is a beta." (The converse will then follow by symmetry.) As before, we rewrite the BNF rule for ⟨alpha⟩ as ⟨alpha⟩ = ⟨digit⟩ $\cup$ ⟨alpha⟩ ⟨digit⟩, which shows us that ⟨alpha⟩ $\subset$ ⟨beta⟩ is equivalent to ⟨digit⟩ $\subset$ ⟨beta⟩ and ⟨alpha⟩ ⟨digit⟩ $\subset$ ⟨beta⟩. The first of these follows immediately from the definition of ⟨beta⟩. As for the second, using our induction principle exactly as before, we find that we may assume that ⟨alpha⟩ ⟨digit⟩ $\subset$ ⟨beta⟩ ⟨digit⟩. However, by the lemma, we know that ⟨beta⟩ ⟨digit⟩ $\subset$ ⟨beta⟩. Therefore ⟨alpha⟩ ⟨digit⟩ $\subset$ ⟨beta⟩, completing the proof.

THE EQUIVALENCE OF KNUTH'S TWO EXAMPLES

Simple context-free induction may easily be extended in two ways: to prove several assertions at once (multiple context-free induction) and to prove assertions about the semantic attributes of a program (semantic induction). As an example of both of these, we shall show the equivalence of the two ways of defining the value of a number as given in [3] to illustrate the ideas of synthesized and inherited attribute.

We shall use a notation for semantic attributes which was first expounded in [6]; this notation is illustrated as follows. Let q^V denote the value of the digit q; let z^V denote the value of the string z of digits, considered as an integer in some base β. The syntactic rule ⟨string of digits⟩ ::= ⟨digit⟩ | ⟨string of digits⟩ ⟨digit⟩ then implies that concat(z, q) (the concatenation of the strings z and q) is also a string of digits. Let us introduce labels into this rule, as follows:

⟨string of digits⟩ x ::= ⟨digit⟩ p; y ::= ⟨string of digits⟩ z ⟨digit⟩ q

In other words, "A string of digits, x, may be a digit p; a string of digits, y, may be a string of digits z followed by a digit q; and these are the only possibilities." Thus concat(z, q) = y, and, if y^V denotes the value of y, then $y^V = \beta \cdot z^V + q^V$ (this is essentially the definition of base-β values). Clearly the value of the string of digits x is just the value of the digit p. These rules define the value of a string of digits as a semantic attribute, in terms of the value of a digit (which is likewise assumed to be a semantic attribute); we may write

⟨string of digits⟩ $\underline{x}$ ::= ⟨digit⟩ $\underline{u}$; $\underline{y}$ ::= ⟨string of digits⟩ $\underline{z}$ ⟨digit⟩ $\underline{q}$
⟨value⟩ $x^V = p^V$; $y^V = \beta \cdot z^V + q^V$

(here ⟨value⟩ is simply a label, giving the intuitive meaning of this attribute). In this notation, Knuth's first example becomes

⟨bit⟩ $\underline{a}$ ::= '0'; $\underline{b}$ ::= '1'
⟨value⟩ $a^V = 0$; $b^V = 1$
⟨list of bits⟩ $\underline{x}$ ::= ⟨bit⟩ $\underline{u}$; $\underline{y}$ ::= ⟨list of bits⟩ $\underline{z}$ ⟨bit⟩ $\underline{v}$
⟨value⟩ $x^V = u^V$; $y^V = 2z^V + v^V$
⟨length⟩ $x^n = 1$; $y^n = z^n + 1$
⟨number⟩ $\underline{x}$ ::= ⟨list of bits⟩ $\underline{a}$; $\underline{y}$ ::= ⟨list of bits⟩ $\underline{b}$ '.' ⟨list of bits⟩ $\underline{c}$
⟨value⟩ $x^V = a^V$; $y^V = b^V + c^V/\exp(2, c^n)$

where exp(p, q) stands for p^q. Thus the value of '1101.01' is 13¼.

Knuth's second example involves an inherited attribute called the scale of a bit or of a list of bits. When a number is of the form L or L.M, where L and M are lists of bits, the scale of L is zero and the scale of M is the negative of its length. For a list of bits of the form B or LB, where B is a bit and L is a sublist, the scale of B is the scale of the entire list, while the scale of L is one more than the scale of the entire list. (Intuitively, the scale of a bit is its distance from the decimal point, which is assumed to be on the right if it is not present.) The value of a one-bit is then 2^s, where s is its scale, and the value of a number is simply the sum of the values of all its component bits. These rules may be expressed as:

⟨bit⟩ $\underline{a}$ ::= '0'; $\underline{b}$ ::= '1'
⟨value⟩ $a^V = 0$; $b^V = \exp(2, b^S)$
⟨scale⟩^{S}
⟨list of bits⟩ $\underline{x}$ ::= ⟨bit⟩ $\underline{u}$; $\underline{y}$ ::= ⟨list of bits⟩ $\underline{z}$ ⟨bit⟩ $\underline{v}$
⟨value⟩ $x^V = u^V$; $y^V = z^V + v^V$
⟨length⟩ $x^n = 1$; $y^n = z^n + 1$
⟨scale⟩^{S}
$u^S = x^S$; $z^S = y^S + 1$; $v^S = y^S$
⟨number⟩ $\underline{x}$ ::= ⟨list of bits⟩ $\underline{a}$; $\underline{y}$ ::= ⟨list of bits⟩ $\underline{b}$ '.' ⟨list of bits⟩ $\underline{c}$
⟨value⟩ $x^V = a^V$; $y^V = b^V + c^V$
$a^S = 0$; $b^S = 0$; $c^S = -c^n$

Here ⟨scale⟩^{S} under ⟨bit⟩ is again simply a label, specifying that a bit has a scale denoted by the superscript s, and similarly for ⟨scale⟩^{S} under ⟨list⟩.

Since the two examples have the same syntax and the same rules for defining length, and since, in fact, the only two corresponding semantic attributes that differ in them are the two notions of value, we shall combine the two definitions into one, and denote the two kinds of value by "value-1" and "value-2":

1. ⟨bit⟩ a ::= '0'; b ::= '1'
2. ⟨value-1⟩ $a^V = 0$; $b^V = 1$
3. ⟨value-2⟩ $a^W = 0$; $b^W = \exp(2, b^S)$
4. ⟨scale⟩^{S}
5. ⟨list of bits⟩ x ::= ⟨bit⟩ u; y ::= ⟨list of bits⟩ z ⟨bit⟩ v
6. ⟨value-1⟩ $x^V = u^V$; $y^V = 2z^V + v^V$
7. ⟨value-2⟩ $x^W = u^W$; $y^W = z^W + v^W$
8. ⟨length⟩ $x^n = 1$; $y^n = z^n + 1$
9. ⟨scale⟩^{S}
10. $u^S = x^S$; $z^S = y^S + 1$; $v^S = y^S$
11. ⟨number⟩ x ::= ⟨list of bits⟩ a; y ::= ⟨list of bits⟩ b '.' ⟨list of bits⟩ c
12. ⟨value-1⟩ $x^V = a^V$; $y^V = b^V + c^V/\exp(2, c^n)$
13. ⟨value-2⟩ $x^W = a^W$; $y^W = b^W + c^W$
14. $a^S = 0$; $b^S = 0$; $c^S = -c^n$

(The numbers 1-14 are for identification only, and are referenced in the proof.)

We wish to prove that the value-1 of any number is the same as its value-2. In order to prove this, we shall have to prove more; namely, that the value-2 of a bit, or of a list of bits, is its value-1 times 2^S, where s is its scale. Formally,

⟨bit⟩ b	15. $b^W = b^V \cdot \exp(2, b^S)$
⟨list of bits⟩ x	16. $x^W = x^V \cdot \exp(2, x^S)$
⟨number⟩ n	17. $n^W = n^V$

The proofs of these are divided into cases, as before. For example, consider the bit '0', denoted by a; we must prove that $a^W = a^V \cdot \exp(2, a^S)$. In fact, this follows from $a^W = 0$ (first part of step 3, above) and $a^V = 0$ (first part of step 2). We will write

$$a^W = 0 = 0 \cdot \exp(2, a^S) = a^V \cdot \exp(2, a^S) \qquad (3a, 2a)$$

Here the first, second, third, etc., alternatives within a rule are denoted by a, b, c, etc. In this style, the proof of statement 15 above proceeds as follows:

⟨bit⟩ a, b:

18. $a^W = 0 = 0 \cdot \exp(2, a^S) = a^V \cdot \exp(2, a^S)$ (3a, 2a)
19. $b^W = \exp(2, b^S) = 1 \cdot \exp(2, b^S) = b^V \cdot \exp(2, b^S)$ (3b, 2b)

The first line here lists the labels (in this case a and b) of the alternative definitions of ⟨bit⟩. The proof of statement 16 is now:

⟨list of bits⟩ x, y:

20. $x^W = u^W = u^V \cdot \exp(2, u^S) = u^V \cdot \exp(2, x^S) = x^V \cdot \exp(2, x^S)$ (7a, 15, 10a, 6a)
21. $y^W = z^W + v^W = z^V \cdot \exp(2, z^S) + v^V \cdot \exp(2, v^S) = z^V \cdot \exp(2, y^S+1) + v^V \cdot \exp(2, y^S)$
$= \exp(2, y^S) \cdot (2z^V + v^V) = y^V \cdot \exp(2, y^S)$ (7b, 16(*), 15, 10b, 10c, 6b)

Here we have used not only steps 6, 7, and 10 of what was given, but also step 15, which has just been proved, as well as step 16 (recursively). Recursive use of a step

is marked by (*). The final conclusion, step 17, is now proved as follows:

⟨number⟩ x, y:

22. $x^w = a^w = a^v \cdot \exp(2, a^s) = a^v = x^v$ (13a, 16, 14a, 12a)

23. $y^w = b^w + c^w = b^v \cdot \exp(2, b^s) + c^v \cdot \exp(2, c^s) = b^v + c^v \cdot \exp(2, -c^n)$
$= b^v + c^v / \exp(2, c^n) = y^v$ (13b, 16, 16, 14b, 14c, 12b)

This completes the proof.

A byproduct of constructions like these, as with any mathematical proof, arises from checking through it to see whether all the assumptions have been used. In this case, nowhere have we ever used step 8 -- in other words, the equivalence of value-1 and value-2 is entirely independent of the definition of the length of a list of bits! Even if we do something completely ridiculous like defining the length of every list of bits to be -4, so that the value-1 of '1101.01' comes out 29 instead of 13¼, we can be assured that the value-2 of '1101.01' will likewise come out 29.

THE VALIDITY OF HOARE'S FIRST AXIOM

We shall now define, syntactically and semantically, a class of programming languages, and show that, for any language in the class, that if the assertion P_0 is true before initiation of the assignment x:=f, then P will be true on its completion, where P_0 is obtained from P by substituting f (actually, in this context, (f)) for all occurrences of x. This is known as the Axiom of Assignment (see Hoare [2]).

Consider the following syntactic rule and associated semantic attribute:

1. ⟨string⟩ x ::= ⟨string element⟩ u; y ::= ⟨string⟩ z ⟨string element⟩ v

2. ⟨substituted string⟩ $x^u(p, q)$ = if u=p then concat('(', q, ')') else u;
$y^u(p, q)$ = concat($z^u(p, q)$, if v=p then concat('(', q, ')') else v)

The substituted string $s^u(p, q)$ is meant to be the result of substituting (q) for p throughout the string s. (Compare the LISP [5] function subst(q, p, s).) We shall be using this definition of the substituted string in investigating the validity of the Axiom of Assignment. First we need a lemma.

LEMMA. If ⟨string⟩ and ⟨substituted string⟩ are defined as above, then

3. ⟨string⟩ x ⟨string⟩ y ⊂ ⟨string⟩ z

4. $z^u(p, q)$ = concat($x^u(p, q)$, $y^u(p, q)$)

That is, the concatenation of any two strings x and y is a string z whose substituted string, for any p and q, is the concatenation of the corresponding substituted strings of x and y.

PROOF. As before, we give the proof as a sequence of steps. This proof differs from the preceding two in that we are proving both a syntactic fact and a semantic fact. Consequently, we use labels (x, y, z, etc.) on the quantities appearing in the syntactic proof, and these labels are then referenced in the semantic proof. We apply

the above definition of a string to the string y; steps 5 and 6 treat the first alternative definition, while steps 7 and 8 treat the second alternative. In each case the string x is appended at the front, just as it appears in step 3 (the syntactic statement to be proved).

5. ⟨string⟩ x ⟨string⟩ y ::= ⟨string⟩ x ⟨string element⟩ e ⊂ ⟨string⟩ z (1a, 1b)

6. $z^u(p, q)$ = concat($x^u(p, q)$, if e=p then concat('(', q, ')') else e)
= concat($x^u(p, q)$, $y^u(p, q)$) (2b, 2a)

7. ⟨string⟩ x ⟨string⟩ y ::= ⟨string⟩ x ⟨string⟩ a ⟨string element⟩ b ⊂ ⟨string⟩ c
⟨string element⟩ b ⊂ ⟨string⟩ z (1b, 3(*), 1b)

8. $z^u(p, q)$ = concat($c^u(p, q)$, if b=p then concat('(', q, ')') else b)
= concat(concat($x^u(p, q)$, $a^u(p, q)$), if b=p then concat('(', q, ')')
else b) = concat($x^u(p, q)$, concat($a^u(p, q)$, if b=p then
concat('(', q, ')') else b)) = concat($x^u(p, q)$, $y^u(p, q)$)(2b,4(*),2b)

The labels in step 5 are referenced in (semantic) step 6; the labels in step 7 are referenced in step 8. Labels are repeated when they refer to the same quantity; thus the string x appears twice in step 5 and twice in step 7, and likewise the string element b appears twice in step 7. The use of (*) to denote induction and of a and b to denote the first and second alternative, respectively, is as before.

We are now ready to state our main result.

THEOREM. Let the following production rules be given as part of the definition of a grammar:

1. ⟨assignment⟩ a ::= ⟨variable⟩ v ':=' ⟨expression⟩ e
2. ⟨effect⟩ $a^e(S) = S'$, where $S'(v) = e^v(S)$, $S'(z) = S(z)$ for $z \neq v$
3. ⟨relation⟩ r ::= ⟨expression⟩ u ⟨relational operator⟩ o ⟨expression⟩ v
4. ⟨value⟩ $r^v(S) = o^o(u^v(S), v^v(S))$
5. ⟨expression⟩ x ::= ⟨term⟩ u; y ::= ⟨adding operator⟩ a ⟨term⟩ v; z ::=
⟨expression⟩ e ⟨adding operator⟩ b ⟨term⟩ w
6. ⟨value⟩ $x^v(S) = u^v(S)$; $y^v(S) = a^u(v^v(S))$; $z^v(S) = b^b(e^v(S), w^v(S))$
7. ⟨term⟩ x ::= ⟨factor⟩ u; y ::= ⟨term⟩ z ⟨multiplying operator⟩ o ⟨factor⟩ v
8. ⟨value⟩ $x^v(S) = u^v(S)$; $y^v(S) = o^o(z^v(S), v^v(S))$
9. ⟨factor⟩ x ::= ⟨constant⟩ k; y ::= ⟨variable⟩ v; z ::= '(' ⟨expression⟩ e ')'
10. ⟨value⟩ $x^v(S) \equiv k^v$; $y^v(S) = S(v)$; $z^v(S) = e^v(S)$
11. ⟨string element⟩ ::= ⟨constant⟩ | ⟨variable⟩ | ⟨adding operator⟩ | ⟨multiplying
operator⟩ | ⟨relational operator⟩ | '(' | ')' | ':='
12. ⟨string⟩ x ::= ⟨string element⟩ u; y ::= ⟨string⟩ z ⟨string element⟩ v
13. ⟨substituted string⟩ $x^u(p, q)$ = if u=p then concat('(', q, ')') else u;
$y^u(p, q)$ = concat($z^u(p, q)$, if v=p then concat('(', q, ')') else v)

Here the following conditions are assumed to hold:

A. The definition of ⟨string⟩ is unambiguous; that is, the eight alternative definitions of a string element are mutually exclusive.

B. Each constant k has a value k^v which is a member of a set D of values.

C. Each adding operator o has an associated unary operator o^u: D → D and an associated binary operator o^b: D × D → D.

D. Each multiplying operator m has an associated binary operator m^o: D × D → D.

E. Each relational operator r has an associated operator r^o: D×D→{true,false}.

F. In the semantic rules, S denotes a state vector, whose domain is the set of all variables and whose range is D, while p denotes a variable and q an expression.

Then Hoare's first axiom holds. Specifically, each relation r is a string whose substituted string is again a relation, and the value of the relation obtained by substituting (q) for p, when applied to an arbitrary state vector S, is the same as the value of the original relation r when applied to the new state vector after the assignment p:=q. Symbolically:

⟨relation⟩ r ⊂ ⟨string⟩ s

$s^u(p, q) \in$ ⟨relation⟩

$(s^u(p,q))^v(S) = r^v((p:=q)^e(S))$

(Note that we must write $s^u(p, q)$ rather than $r^u(p, q)$, since the production rules 1-13 above do not define the substituted string of a relation.)

Before proving the theorem, we examine its form and a few generalizations.

The syntactic and semantic rules given in the statement of the theorem constitute a programming language schema, or abstraction of the concept of a programming language, since the form of the constants, variables, and operators is left unspecified. Such a schema resembles the "abstract syntax" of McCarthy [4], except that McCarthy also leaves unspecified the order of the components of each alternative, whereas we, for example, have required that all our binary operators be infix. The need for a slightly lower level of abstraction may be seen by noting that there exist properties of programming languages which do, in fact, depend on whether operators are prefix, infix, or suffix, although not necessarily on the form of the terminals.

The current state of the computation is represented by a state vector S, viewed as a function from variables to their current values. Expressions, terms, factors, and relations are presumed to have values which are dependent upon the current state vector; an assignment has an effect, which is a function giving the new state vector after the assignment in terms of the current state vector before the assignment. The values of a relation are true and false, depending on whether that relation is or is not (currently) satisfied; the values of expressions, terms, and factors are presumed to be drawn from the same set D as are the values of constants.

Since a^e denotes the effect of the assignment a, $(p:=q)^e$ denotes the effect of the specific assignment p:=q where p is a variable and q is an expression. (See [6].)

The relation of our symbolic statement of Hoare's first axiom to its usual statement is as follows. If $P^v(S)$ denotes the current value of the predicate P when the current state vector is S, while $Q^e(S)$ denotes the next state vector after the execution of Q with current state vector S, then Hoare's first axiom is equivalent to "If $P^v(S)$ = true, then $R^v(Q^e(S))$ = true," where R is our relation r, which is a

string s; P is the substituted string $s^u(p, q)$; and Q is the assignment p:=q. We actually prove the stronger statement $P^v(S) = R^v(Q^e(S))$; in other words, R is true after Q is executed if and only if P is true beforehand.

The theorem suggests a number of generalizations. The form of a relation may be generalized; exponential and other operators may be introduced; even side effects may be introduced, as long as they are carefully controlled. We have purposely not attempted to define a "most general" language schema for which Hoare's first axiom holds, and indeed it is doubtful whether such a schema exists. Our proof method is readily susceptible to computer-aided treatment, allowing us to construct and test syntactic and semantic definitions of subsets of FORTRAN, ALGOL 60, and so forth.

PROOF. As before, we shall need to prove more than what is stated. In fact, we shall prove that the given assertion about the values of a relation before and after an assignment holds for expressions, terms, and factors, as well as relations. We lay out the statements to be proved, as follows:

14. ⟨factor⟩ f ⊂ ⟨string⟩ s
15. $s^u(p, q) \in$ ⟨factor⟩
16. $(s^u(p, q))^v(S) = f^v((p:=q)^e(S))$
17. ⟨term⟩ t ⊂ ⟨string⟩ s
18. $s^u(p, q) \in$ ⟨term⟩
19. $(s^u(p, q))^v(S) = t^v((p:=q)^e(S))$
20. ⟨expression⟩ e ⊂ ⟨string⟩ s
21. $s^u(p, q) \in$ ⟨expression⟩
22. $(s^u(p, q))^v(S) = e^v((p:=q)^e(S))$
23. ⟨relation⟩ r ⊂ ⟨string⟩ s
24. $s^u(p, q) \in$ ⟨relation⟩
25. $(s^u(p, q))^v(S) = r^v((p:=q)^e(S))$

Here, as before, p is an arbitrary variable, q an expression, and S a state vector.

The proof now proceeds as follows:

26. ⟨factor⟩ x ::= ⟨constant⟩ k ⊂ ⟨string element⟩ e ⊂ ⟨string⟩ s (9a, 11a, 12a)
27. $s^u(p, q)$ = (if k=p then concat('(', q, ')') else k) = k = x ∈ ⟨factor⟩ (13a,A)
28. $(s^u(p, q))^v(S) = x^v(S) = k^v = x^v((p:=q)^e(S))$ (27, 10, 10)
29. ⟨factor⟩ y ::= ⟨variable⟩ v ⊂ ⟨string element⟩ e ⊂ ⟨string⟩ s (9b, 11b, 12a)
30. $s^u(p, q)$ = (if v=p then concat('(', q, ')') else v) ∈ ⟨factor⟩ (13a, 9c, 9b)
31. $(s^u(p, q))^v(S)$ = (if v=p then $(\text{concat}('(', q, ')'))^v(S)$ else $v^v(S)$) = (if v=p then $q^v(S)$ else S(v))= $((p:=q)^e(S))(v) = y^v((p:=q)^e(S))$ (13a, 10c, 10b, 2, 10b)
32. ⟨factor⟩ z ::= '(' ⟨expression⟩ e ')' ⊂ ⟨string element⟩ a ⟨string⟩ b ⟨string element⟩c⊂⟨string⟩ u ⟨string⟩v⊂⟨string⟩s (9c, 11f, 20(*), 11g, 12a, 12b, LEMMA)
33. $s^u(p, q)$ = concat($u^u(p, q)$, $v^u(p, q)$) = concat(if a=p then concat('(', q, ')') else a, concat($b^u(p, q)$, if c=p then concat('(', q, ')') else c)) = concat('(', concat($b^u(p, q)$, ')')) ∈ '(' ⟨expression⟩ e ')' ⊂ ⟨factor⟩ (LEMMA, 13a, 13b, A, A, 21(*), 9c)
34. $(s^u(p, q))^v(S) = (\text{concat}('(', \text{concat}(b^u(p, q), ')')))^v(S) = (b^u(p, q))^v(S) = e^v((p:=q)^e(S)) = z^v((p:=q)^e(S))$ (33, 10c, 22(*), 10c)

(This completes the analysis for statements 14, 15, and 16)

35. ⟨term⟩ x ::= ⟨factor⟩ u ⊂ ⟨string⟩ s (7a, 14)

36. $s^u(p, q) \in$ ⟨factor⟩ ⊂ ⟨term⟩ (15, 7a)

37. $(s^u(p, q))^v(S) = u^v((p:=q)^e(S)) = x^v((p:=q)^e(S))$ (16, 8a)

38. ⟨term⟩ y ::= ⟨term⟩ z ⟨multiplying operator⟩ o ⟨factor⟩ v ⊂ ⟨string⟩ a ⟨string element⟩ b ⟨string⟩ c ⊂ ⟨string⟩ d ⟨string⟩ c ⊂ ⟨string⟩ s (7b, 17(*), 11d, 14, 12b, LEMMA)

39. $s^u(p, q) = concat(d^u(p, q), c^u(p, q)) = concat(concat(a^u(p, q),$ if o=p then concat('(',q,')') else o),$c^u(p, q)) = concat(concat(a^u(p, q), o), c^u(p, q)) \in$ ⟨term⟩⟨multiplying operator⟩⟨factor⟩ ⊂ ⟨term⟩ (LEMMA,13b,A,18(*),15,7b)

40. $(s^u(p, q))^v(S) = (oconcat(concat(a^u(p, q), o), c^u(p, q)))^v(S) = o^o((a^u(p, q))^v(S), (c^u(p, q))^v(S)) = o^o(z^v((p:=q)^e(S)), v^v((p:=q)^e(S))) = y^v((p:=q)^e(S))$ (39, 8b, 19(*), 16, 8b)

(This completes the analysis for statements 17, 18, and 19)

41. ⟨expression⟩ x ::= ⟨term⟩ u ⊂ ⟨string⟩ s (5a, 17)

42. $s^u(p, q) \in$ ⟨term⟩ ⊂ ⟨expression⟩ (18, 5a)

43. $(s^u(p, q))^v(S) = u^v((p:=q)^e(S)) = x^u((p:=q)^e(S))$ (19, 6a)

44. ⟨expression⟩ y ::= ⟨adding operator⟩ a ⟨term⟩ v ⊂ ⟨string element⟩ b ⟨string⟩ d ⊂ ⟨string⟩ c ⟨string⟩ d ⊂ ⟨string⟩ s (5b, 11c, 17, 12a, LEMMA)

45. $s^u(p, q) = concat(c^u(p, q), d^u(p, q)) =$ concat(if a=p then concat('(', q, ')') else a, $d^u(p, q)) = concat(a, d^u(p, q)) \in$ ⟨adding operator⟩ ⟨term⟩ ⊂ ⟨expression⟩ (LEMMA, 13a, A, 18, 5b)

46. $(s^u(p, q))^v(S) = (concat(a, d^u(p, q)))^v(S) = a^u((d^u(p, q))^v(S)) = a^u(v^v((p:=q)^e(S))) = y^v((p:=q)^e(S))$ (45, 6b, 19, 6b)

47. ⟨expression⟩ z ::= ⟨expression⟩ e ⟨adding operator⟩ b ⟨term⟩ w ⊂ ⟨string⟩ a ⟨string element⟩ c ⟨string⟩ d ⊂ ⟨string⟩ g ⟨string⟩ d ⊂ ⟨string⟩ s (5c, 20(*), 11c, 17, 12b, LEMMA)

48. $s^u(p, q) = concat(g^u(p, q), d^u(p, q)) = concat(concat(a^u(p, q),$ if b=p then concat('(', q, ')') else b), $d^u(p, q)) = concat(concat(a^u(p, q), b), d^u(p, q)) \in$ ⟨expression⟩ ⟨adding operator⟩ ⟨term⟩ ⊂ ⟨expression⟩ (LEMMA, 13b, A, 21(*), 18, 5c)

49. $(s^u(p, q))^v(S) = (concat(concat(a^u(p, q), b), d^u(p, q)))^v(S) = b^b((a^u(p, q))^v(S), (d^u(p, q))^v(S)) = b^b(e^v((p:=q)^e(S)), w^v((p:=q)^e(S))) = z^v((p:=q)^e(S))$ (48, 6c, 22(*), 19, 6c)

(This completes the analysis for statements 20, 21, and 22)

50. ⟨relation⟩ r ::= ⟨expression⟩ u ⟨relational operator⟩ o ⟨expression⟩ v ⊂ ⟨string⟩ a ⟨string element⟩ b ⟨string⟩ c ⊂ ⟨string⟩ d ⟨string⟩ c ⊂ ⟨string⟩ s (3, 20, 11e, 20, 12b, LEMMA)

51. $s^u(p, q)$ = concat($d^u(p, q)$, $c^u(p, q)$) = concat(concat($a^u(p, q)$, if o=p then concat('(', q, ')') else o), $c^u(p, q)$) = concat(concat($a^u(p, q)$, o), $c^u(p, q)$) expression relational operator expression relation (LEMMA, 13b, 21, 21, 3)

52. $(s^u(p, q))^v(S)$ = $(\text{concat}(\text{concat}(a^u(p, q), o), c^u(p, q)))^v(S)$ = $o^o((a^u(p, q))^v(S), (c^u(p, q))^v(S))$ = $o^o(u^v((p:=q)^e(S)), v^v((p:=q)^e(S)))$ = $r^v((p:=q)^e(S))$ (51, 4, 22, 22, 4)

(This completes the analysis for statements 23, 24, and 25) Q. E. D.

A few remarks are in order about the form of the proof. The statement A in the hypothesis of the theorem is used during the proof to show that expressions of the form if x=p then concat('(', q, ')') else x must in fact be equal to x if x is a constant (step 27), an adding operator (steps 45 and 48), a multiplying operator (step 39), a relational operator (step 51), or a left or right parenthesis (step 33). The reason, of course, is that "x=p" must be false, since p represents a variable and statement A says that variables and the other quantities mentioned here are distinct. It is also necessary to note carefully that variables, and not the individual characters in variable identifiers, are taken as elements of strings. If we were to treat characters as string elements, the definition of the substituted string would have to be considerably more complex, because we would have to be careful not to replace substrings of an identifier. We do not wish to change "alpha" into "al(q)ha" when substituting (q) for p, for example.

ACKNOWLEDGMENT

This research was partially supported by National Science Foundation Grant GJ-31612. The author is grateful to Ralph London, Michael Megas, Nori Suzuki, and Raymond Wong for their helpful comments and suggestions.

REFERENCES

1. Burstall, R. M., Proving properties of programs by structural induction, Computer J. 12, 41-48 (1969).

2. Hoare, C. A. R., An axiomatic basis for computer programming, Communications of the ACM 12, 576-580 and 583 (1969).

3. Knuth, D. E., Semantics of context-free languages, Math. Systems Theory 2, 127-145 (1968).

4. McCarthy, J., Towards a mathematical science of computation, Information Processing 1962, Proc. of IFIP Congress 62, North-Holland, Amsterdam, 21-28 (1963).

5. McCarthy, J., et al., LISP 1.5 Programmer's Manual, MIT Press (1962).

6. Maurer, W. D., A semantic extension of BNF, International J. of Computer Math., Section A, Vol. 3, 157-176 (1972).