MAC TR-124

VERIFICATION OF PROGRAMS OPERATING ON STRUCTURED DATA

Mark Steven Laventhal

This research was supported by the National Science Foundation under research grant GJ-34671.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

VERIFICATION OF PROGRAMS OPERATING ON STRUCTURED DATA

by

Mark Steven Laventhal

Submitted to the Department of Electrical Engineering on January 23, 1974 in partial fulfillment of the requirements for the Degrees of Bachelor of Science and Master of Science.

ABSTRACT

The major method for verifying the correctness of computer programs is the inductive assertion approach. This approach has been limited in the past by the lack of techniques for handling data structures. In particular, there has been a need for concepts with which to describe structured data during intermediate and final stages of a computation. This thesis describes an approach by which this problem can be handled, and demonstrates its use in proving several programs correct.

The key to the approach is the restriction of a data structure to a particular structural class. Primitive concepts are introduced which allow such a class to be concisely defined. Other concepts relate structures from a given class to data abstractions which the structures can be thought to represent. It is shown how to integrate the structural descriptions with the actual proofs of correctness by incorporating results of general applicability into a logical formalism for a given structural class.

THESIS SUPERVISOR: Barbara H. Liskov
TITLE: Assistant Professor of Electrical Engineering

ACKNOWLEDGEMENTS

I wish to express my appreciation to Professor Barbara Liskov for patiently supervising this research, and in particular for helping me select the specific topic of this thesis and then several times preventing me from wandering too far afield.
I also want to thank Bill Mark and Jack Aiello for reading previous drafts of the thesis, aiding me considerably both by their general observations and with detailed proofreading.
Finally, I wish to acknowledge the National Science Foundation Graduate Fellowship Program for its financial support during the time I have been engaged in this research.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION TO PROGRAM VERIFICATION

One of the first and saddest facts which a beginning computer programmer learns is that programs do not always do what they are supposed to. By this, I do not refer to the fact that a misplaced semicolon can mean the difference between a perfectly operating program and one which does not make it past the parsing stage of a compiler. What I mean is that a syntactically correct, fully compiled program will be fed some input data, run to completion, and produce a result different from the one the programmer desired.

The first reaction of the novice programmer to this situation may well be to question the reliability of the computer upon which his program is being run. If wiser and more experienced heads are around, however, they will persuade the beginner to "hand-simulate" his program on the given input data to determine the nature of the error. In doing so, he will laboriously repeat the computation performed by the computer, presumably arriving at the same answer which it provided. While he will now see that the computer is correct, he may not yet see what in the program is wrong. He may then be counselled to go back and hand-simulate the program again, this time on general input data, however, keeping all

computations in symbolic form. In doing so, he may discover that his DO-loop is always executed once more than he wants. Upon making the appropriate change, he is able to arrive at his first correctly-functioning computer program.

What knowledge has the programmer gained from this episode? One hopes it is more than just to be careful about the range of a DO-loop. What he should learn from this experience is that hand-simulation of a program on symbolic data is one way to debug, and conversely to verify the correctness of, a program. Perhaps the next time he writes a program, before submitting it for a test run he will not only check it over for syntax errors, but also attempt to verify its correctness using this technique.

A. Inductive assertions

The desirability of proving a program correct was recognized soon after the concepts of the stored-program computer and programming itself were conceived. Goldstine and von Neumann [Gol63] proposed a method of verifying correctness not very different from the informal notion of hand-simulation featured in the opening vignette. Their method used the concept of a state vector, a list of program variables with their corresponding values (in symbolic form) at a given control point in the program. Their idea was that the programmer could provide a symbolic state vector presumed to correspond to the control

point following each statement in his program. The verification procedure would then consist of confirming for each statement numbered i in the program: that the validity of the state vector following statement (i - 1) prior to the execution of statement i ensures the validity of the state vector following statement i after its execution. Proving this for each statement in the program would result in a proof that the computation of the program as a whole corresponds to the state vector following the last program statement.

This procedure, while not terribly profound, is only a less sophisticated version of the best approach at present for proving a program's correctness. The one technique with widespread use, and applicable to programs with both applicative and imperative linguistic features, is the inductive assertion approach, first proposed by Floyd[Flo67] and Naur[Nau66] . Other techniques, such as recursion induction[McC63] and structural induction[Bur69], are limited to programs written in a purely applicative language like pure LISP or R-PAL[Woz71].[1]

At the heart of the assertion approach are the assertions themselves. These are Boolean expressions which characterize some or all of the data on which the program operates. An assertion is attached to a control point in

[1]For a discussion of several of these other techniques, see Greif[Gre72].

the program, either a line of source code or an edge of a flowchart; in either case, the assertion is alleged to be true each time control reaches that point during the execution of the program. Assertions are thus the natural generalization of state vectors. Instead of specifying a vector of values, an assertion allows a virtually free-form description of the data in the program. The one constraint is that the assertion as a whole be a Boolean expression. This is one important advantage of using assertions rather than state vectors.

The other major improvement of the assertion approach over the idea of Goldstine and von Neumann is that assertions need not be attached to every program statement. In fact, an inductive assertion proof requires only an input assertion, an output assertion, and one assertion for each loop in the program. To simplify a given proof, of course, additional intermediate assertions may be helpful and thus included, but one need provide no more than this minimum. Since the actual construction of the assertions is the most difficult aspect of the method (as noted by Good[Goo70] and Deutsch[Deu73], among others), this savings is extremely significant. In addition, the inductive character of the proof technique (see below) allows more complicated iterative control structures to be handled.

The input assertion of a program, also called the antecedent or precondition, usually specifies the domains of, and any joint constraints among, the input variables. The output assertion, also called the consequent or post-condition, generally expresses the desired result of the program. The "proof of correctness" or "verification" is actually a proof that if the input assertion is true upon entry to the program, then the program will exit with its output assertion true. The loop assertions are required to construct the proof, as explained below. It is thus important that all assertions, particularly the antecedent and consequent, be stated correctly. An erroneous assertion can cause the proof of a correct program to fail, or that of an incorrect one to succeed.[1]

The assertions are sometimes called "inductive" assertions because the proof of the program's correctness is by induction on the number of iterations through its loops. For example, consider a program with the general control structure given by the macro-flowchart in Figure 1.1. INITIALIZATION, LOOP-BODY and FINALIZATION are intended to be loop-free program fragments, and LOOP-TEST is a simple predicate whose truth denotes the condition for loop termination. P, Q and R are assertions

[1] As long as the antecedent and consequent are correct, however, an improper intermediate assertion can never allow an incorrect program to be "verified."

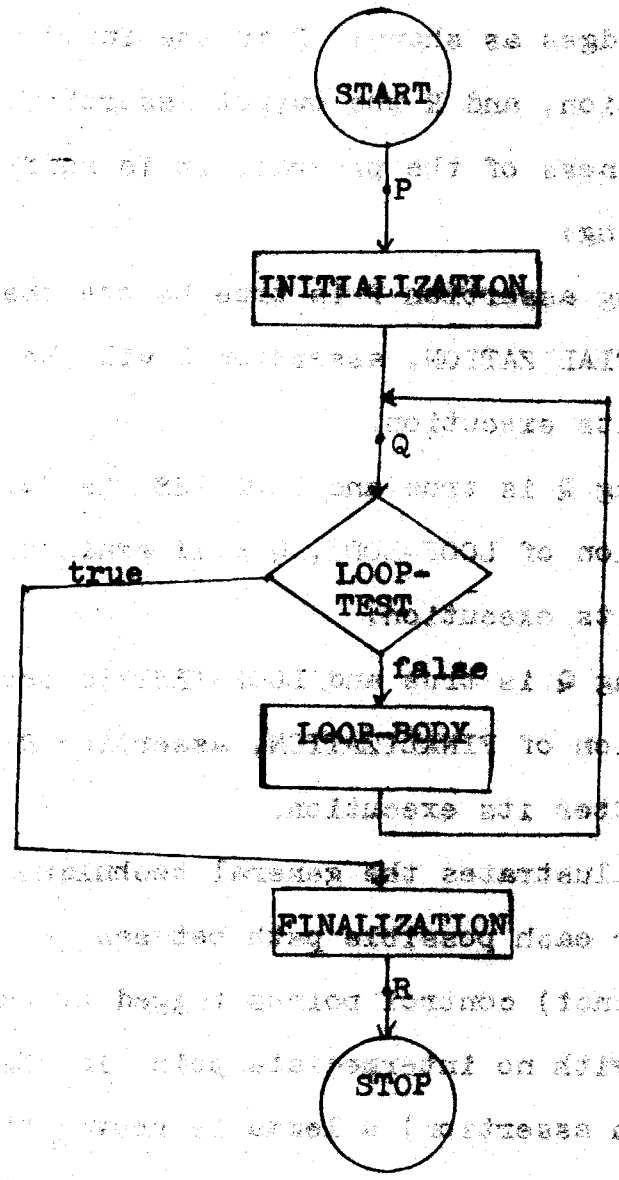


Figure 1.1

Control structure for a single-loop program

attached to the edges as shown: P is the input assertion, Q the loop assertion, and R the output assertion. To prove the correctness of the program, it is sufficient to prove the following:

(1) Assuming assertion P is true before the execution of INITIALIZATION, assertion Q will be true after its execution.

(2) Assuming Q is true and LOOP-TEST is false before execution of LOOP-BODY, Q will again be true after its execution.

(3) Assuming Q is true and LOOP-TEST is true before execution of FINALIZATION, assertion R will be true after its execution.

The above illustrates the general technique for assertion proofs. For each possible path between two (not necessarily distinct) control points tagged by assertions, say P1 and P2, (with no intermediate point in the path also tagged by an assertion) a lemma is proved that if the path is taken with assertion P1 true at the first control point, then assertion P2 will be true when the second control point is reached. If each such lemma can be proved, then the proof of the theorem that the entire program is correct is a simple induction argument.

There is one flaw in the above argument: it implicitly assumes that each loop will eventually terminate.

However, there is no assurance that in the example above, say, LOOP-TEST will not always evaluate to false, resulting in an infinite loop and a non-terminating program. The above type of proof, then, is only of *partial correctness* [Man69]; it proves that *if* the program terminates, the result will be correct. To complete the proof, one must prove in addition that each loop terminates.

There are several bits of nomenclature associated with inductive assertion proofs. The lemmas to be proved for the various paths within the program are called *verification conditions* (*v.c.'s* for short), since they represent the conditions sufficient for verifying the overall correctness of the program. Within a given v.c., the assertions tagging its beginning and ending points are called the *initial assertion* and *final assertion* of the v.c., respectively. The source code in the path between these two endpoints is called the *body* of the v.c. Because each loop in the program is cut by an assertion, the body of a v.c. is guaranteed to be loop-free.

Nothing in this mechanism is in any way dependent on the code under consideration being a "program." In fact, the technique is equally applicable for typeless procedures, functions, and even mere sections of code. In this thesis, the computational objects used in proofs will be procedures in an ALGOL-like language (the details of which will be discussed in Chapter 2).

B. Example proof using assertions

A more concrete example should serve to clarify these ideas. Consider the simple procedure given in Figure 1.2. Its purpose is to divide positive integer y into positive integer x, giving quotient q and remainder r. (It is taken from Floyd's original paper on proving programs via assertions [Flo67].) The assertions are attached to control points in the program by including them as comments located at the corresponding points in the source code. Because there is one loop in the program, there are three assertions—the precondition, the postcondition, and the loop assertion.

A total of four verification conditions must be proved, corresponding to the four possible paths between points tagged by assertions. The first v.c. is that the truth of the input assertion followed by the execution of lines (1) through (4) results in the truth of the loop assertion. This can be abbreviated[1]:

$$(x > 0 \ \underline{\text{and}} \ y > 0) \ \{(1),(2),(3),(4)\} \ (0 < x \ \underline{\text{and}} \ x = r + y*q \ \underline{\text{and}} \ 0 < y \leq r).$$

Its proof goes like this:

(a) Since "x > 0" is in the initial assertion of the v.c. and the value of x is unchanged by the

[1] In general, "P {Q} R" is a notational shorthand for: "If the body of code given or represented by Q is executed with assertion P true as the precondition, then postcondition R will be true." Hoare [Hoa69] is the inventor of this notation.

```
     procedure divide(x,y,q,r);
          declare x,y,q,r integer;
          begin
          comment assert "x ≥ 0  and  y > 0";
(1)       r := x;
(2)       q := 0;
(3)       while  y ≤ r  do
(4)            begin
               comment assert "0 ≤ x = r + y*q  and
                                    0 < y ≤ r";
(5)            r := r - y;
(6)            q := q + 1;
(7)            end;
          comment assert "0 ≤ x = r + y*q  and
                                 0 ≤ r < y";
          end divide;
```

Figure 1.2

Example program

action of the body, "x > 0" must be valid in the final assertion. Similarly for "y > 0".

(b) The action of line (1) assigns to r the value of x. Since this is the only assignment to r in the body, r = x at the end of the body. Similarly, q = 0 from line (2). Thus, r + y*q = x + y*0 = x, so "r + y*q = x" is valid in the final assertion.

(c) Since the loop body is entered, it follows that the test "y $\leq$ r" on line (3) must evaluate to <u>true</u>. Thus, since neither variable is changed between line (3) and the end of the body, "y $\leq$ r" must be valid finally.

This proves the validity of the entire final assertion, assuming the initial validity of the initial assertion. The v.c. is therefore verified.

The second v.c. is that the input assertion followed by the action of lines (1) and (2) and a <u>false</u> outcome of the test at line (3) ensure the final validity of the output assertion. The proof of this v.c. proceeds identically to that of the first, the only difference being in part (c) of the proof. Here,

(c) Since the loop body is not entered, it follows that the test "y $\leq$ r" on line (3) must evaluate to <u>false</u>. Thus, since neither variable

gets changed between line (3) and the end of the body of the v.c., $r \leq y$. Furthermore, since $r = x$ and $x > 0$, it must be that $r \geq 0$, so "$0 \leq r < y$" must be finally valid.

The third verification condition is that one iteration through the loop body preserves the validity of the loop assertion, that is:

$$(0 < x = r + y*q \text{ and } 0 < y \leq r) \ \{(5),(6),(7),(3),(4)\} \ (0 < x = r + y*q \text{ and } 0 < y \leq r).$$

Its proof:

(a) Since "$0 < x$" and "$0 < y$" are valid initially, and both x and y are unchanged by the body, both clauses are valid after execution of the body.

(b) Denote the new values of r and q by r' and q', and the old values by r_i and q_i. Then by the actions of lines (5) and (6) respectively, $r' = r_i - y$ and $q' = q_i + 1$. So,

$$\begin{aligned} r' + y*q' &= (r_i - y) + y * (q_i + 1) \\ &= r_i - y + y*q_i + y \\ &= r_i + y*q_i \end{aligned}$$

which by the initial assertion of the v.c. is equal to x. Thus, "$r + y*q = x$" is valid in the final assertion.

(c) Since the loop body is re-entered, it follows

that the "y $\leq$ r" test on line (3) must be <u>true</u>, and since neither y nor r is subsequently changed, "y $\leq$ r" in the final assertion follows.

The last verification condition is:

$(0 < x = r + y*q$ <u>and</u> $0 < y \leq r)$ (5),(6),(7),(3) $(0 < x = r + y*q$ <u>and</u> $0 \leq r < y)$

where the test on line (3) has an outcome of <u>false</u>.

(a) "$0 < x = r + y*q$" in the final assertion follows from its validity in the initial assertion by reasoning similar to that used in parts (a) and (b) in the third v.c.

(b) Since the loop is exited, the "y $\leq$ r" test on line (3) must have been <u>false</u>, so "$r < y$" follows. Since "y $\leq$ r" is true in the initial assertion and r is decremented by the value of y on line (5) and unchanged otherwise, we have that $0 \leq r$, so $0 \leq r < y$.

Finally, we must prove that the loop does, in fact, terminate. The termination condition, by line (3), is "$r < y$". By line (1), r is initialized to the value of x, which the input assertion states to be a finite, positive quantity. Each iteration of the loop decreases r by the value of y (on line (5)), also a finite, positive quantity (by the loop assertion). It is thus an easy matter to see

that (assuming the initial validity of the input assertion) eventually "r < y" is true and the loop terminates.

C. Symbolic interpretation

In proving the verification conditions above, a technique is employed which Burstall calls symbolic interpretation[Bur72b]. The basis of this approach is the method of "hand-simulation" mentioned earlier. That is, the person proving the v.c. "acts" as the computer, interpreting the source code in the body of the v.c. This "interpretation," however, is carried out in symbolic form. The initial values of all variables are unknown, except as they may be constrained by the initial assertion.

As Good [Goo70] shows, there are several variations under the general heading of "symbolic interpretation". Good discusses and demonstrates the equivalence of a few of these: forward accumulation, forward substitution, and backward substitution. The differences among these lie in whether one works backward or forward through the body of code, and whether one accumulates new results in separate instantiations of a variable(e.g. x_1, x_2, x_3, ...) or substitutes the new results back into a single instance (i.e., simply x). Which of these variations one employs is chiefly a matter of taste and convenience; both forward and backward substitution are used in this thesis. In its various forms, symbolic interpretation is the most commonly used technique for proving verification conditions.

The major alternative to this method is the formal approach taken by Hoare. This approach uses the more mathematically rigorous concepts of axioms and rules of inference to effect proofs. Each axiom corresponds to one kind of simple statement, and each rule of inference to a feature of control structure. A proof is then generally built from the bottom up, starting with proofs of the effects of individual statements by direct application of axioms, and proceeding by combining these proofs into proofs of progressively larger program fragments, until ultimately the entire proof has been constructed. In his first major paper explaining his approach [Hoa69], Hoare supplied an axiom for simple ALGOL assignment statements and rules of inference to cover statement composition, conditional statements, and iteration. With these basic tools and elementary axioms of arithmetic, he proved the "divide" procedure of Figure 1.2 in twelve steps. In subsequent work, Hoare has extended his rules of inference to include function calls and a restricted form of jump [Cli72], and procedure calls [Hoa71].

The major drawback to Hoare's approach, as I see it, is that it is simply not a "natural" way for humans to prove programs. Eventually, when mechanization of the proof process on the computer becomes critical, the rela-

tive informality of symbolic interpretation may have to give way to the rigor of Hoare's approach (although not necessarily; Good[Goo70], King[Kin69], Deutsch[Deu73] and others have had success in automating proofs using symbolic interpretation). At the moment, though, program verification is still in its formative stages, and as Burstall has noted[Bur69]:

> 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.

One outgrowth of Hoare's work is an excellent notation for expressing the semantics of various source-language features, a notation which is employed later in this thesis.

D. Data structures in assertion proofs

As Deutsch [Deu73] has observed, one difficulty with the assertion approach in practice is that it requires the semantics of the source language be fully understood and expressible. Hoare has shown that by excluding the unrestricted jump, all major control structures can be handled. Since the "goto" statement has been claimed to be harmful (Dijkstra[Dij68], etc.) and in any case, been proven to be unnecessary (e.g., see Ashcroft and Manna[Ash71]), it is clear that control structures present no major obstacles to proving programs correct.

Unfortunately, the area of structured data did not at first receive the same degree of attention as that of control structures. Consequently, work in this area has tended to lag behind, and only recently has there been much done to catch up. The first programs to be verified operated only on simple integer data. Obvious extensions to other simple data classes(real numbers, Booleans, etc.) followed, and more importantly, techniques to handle arrays were developed([Goo70], [Kin69]). The work with arrays constituted the first involvement of verification with structured data, though the structuring was simple and inflexible.

In the past year or so, several significant works have emerged on more complicated data structures. This includes the work by Burstall[Bur72a] and by Poupon and Wegbreit[Pou72] on LISP-type lists, and that of Morris [Mor72] on data structures with pointers. The latter two works typify what I call the generalist approach. The generalist deals with a class of data structures general enough to include (disregarding nomenclature) all possible data structures. Both of the works cited obtain some useful results, but neither is developed sufficiently to provide a complete framework for correctness proofs. Instead, each provides some tools which are useful but insufficient by themselves.

The alternative is to restrict the class of data structures under consideration in order to derive a complete (or nearly so) treatment with, of course, less generality of scope. Burstall [Bur72a] considers a class of LISP-type lists which he calls "Distinct Non-repeating List Systems." By the elimination of nesting of lists and the restriction of cycles, Burstall is able to obtain a quite satisfactory overall system for proofs of correctness. The challenge, of course, once such a system is developed, is to progressively expand its domain of application by successive weakening and/or elimination of the restrictions.

E. Approach of this thesis

It is my contention that the most promising approach is a compromise between these two schools of thought. In particular, I believe that rather than develop specific new techniques and concepts, work in this area should strive to devise new classes of such tools. The specific tools to be used for an actual application can then be designed by tailoring specific elements of these classes for use in the given application.

The primary emphasis in this thesis, then, is to develop general techniques for proving the correctness of programs involving data structures. This will entail considering the way data structures are used in programs

and the types of characterizations of structures which are necessary for constructing relevant assertions. New concepts will be devised which are directly translatable into the tools required for proofs of correctness. This area will be the focus of Chapter 3.

The development of new general concepts, however, without a detailed instantiation to demonstrate their practicality and utility is highly suspect. It therefore becomes incumbent upon me to illustrate the use of the techniques I describe by actually verifying programs which operate on data structures. The tools designed in Chapter 3 are intended to be specifically tailored for use with a particular class of structures. My demonstration is therefore on a specific structural class—simple singly-linked lists—and is presented in Chapter 4. This data class has been chosen primarily for its simplicity, in order to keep extraneous details and attendant "hair" to a minimum. The example programs proved in Chapter 4, however, solve real-world problems, and the proofs are non-trivial enough to show the general utility of the approach.

Before the substantive issues of this work can be tackled, though, an important administrative detail must be handled. In order for the rest of this thesis to be completely clear, the reader must be able to understand

the source language I am using. While most features of the language are obvious, those aspects relating to data structures have been specially designed and therefore require explicit explanation. This explanation is provided in Chapter 2.

Chapter 2

DATA STRUCTURING IN THE SOURCE LANGUAGE

The source language used throughout this thesis is a simple ALGOL-like language. Aside from its data-structuring facilities, which are based on Hoare records [Hoa68], it is effectively a subset of ALGOL W [Wir66]. Certain features of ALGOL W have been deliberately excluded, such as labels and jumps, but defining the precise extent of the language is not important. The aspects relating to data structures are carefully explained in this chapter. The rest of the language is intended to be obvious in both its syntax and semantics. For any detail which is not covered in this chapter explicitly, the ALGOL W paper [Wir66] can be used as a "reference manual."

A. Record classes

The record-handling facility of the language allows compound data objects to be composed from a group of simple objects. Arrays perform a similar function, of course, but the individual elements of a record need not all be of the same data type. As shall be seen shortly, it is this property which allows interesting data structures to be constructed using records.

A record class definition does not itself create any data object. Rather, it establishes a new data type of

which data objects can be created. The record class definition provides the pattern for a typical record of the class. Each component element of the typical record is described by giving its data type and a unique identifier, called a selector, for accessing it.

The syntax for a record class definition is given by the following set of BNF production rules:

<record class definition> ::=

record class <identifier> (<body>)

<body> ::= <components> | <components>; <body>

<components> ::= <id list> <data type>

<id list> ::= <identifier> | <identifier>, <id list>

An example of a record class definition is:

record class rc(x,y,z real; sw boolean;

table integer array [1:20])

Each record of class "rc" would then contain five components: three real components, selected by identifiers x,y and z; one Boolean component selected by sw; and a 20-element integer array component selected by table.

B. Reference variables and component selection

The definition of record class "rc" establishes a new data type: "reference(rc)", which can be shortened to "ref(rc)". A variable of a reference type either references a record of the given record class or possesses

the special value null. All reference variables are automatically initialized to null on being declared. A reference variable can refer to any record of the one class, but not to any record of any other record class.

A component of a record is accessed by an expression of the form

"⟨reference expression⟩.⟨selector identifier⟩".

The ⟨reference expression⟩ is an expression which evaluates to a reference to the given record, and the ⟨selector identifier⟩ selects the specific component of the record. So, if variable r is declared to be of type ref(rc), then "r.x", "r.y", "r.z", "r.sw", and "r.table" are all examples of component selection. The reference expression may itself contain a component selection in the case of a reference-typed record component. As the above indicates, the selection operation "." associates to the left.

It should be noted that the type of ⟨reference expression⟩ can always be determined statically, i.e. at a hypothetical "compilation time." A compiler could therefore ensure that the expression references a record of a class for which the selector is valid, or else register a compilation error. In verifying the correctness of programs, we are solely concerned with dynamic correctness. The program to be verified is assumed to have successfully compiled, and all static issues can thus

be ignored. The record-handling facility described here, allowing for a maximum of type-checking, frees us to as great an extent as possible from having to worry about these issues. It is with this in mind that the data-structuring aspects of the language have been designed.

While the class of the record to which ⟨reference expression⟩ refers can be determined statically, there is a related issue which can only be checked dynamically. If the expression evaluates to null, then the component selection is undefined, which causes the program not to terminate normally. In proving a program correct, all component selection must be shown to be well-defined, in order that total correctness may be deduced from the proof of partial correctness.

The data type of a component-selection expression is simply the declared type of the selected component. Such an expression can occur in any context which is appropriate for a variable of that type. This includes the left-hand-side of an assignment statement—that is, individual components of a record are individually updatable. In fact, once a record has been created, the only way to change the values of its components is by updating individual components.

The left-hand-side of an assignment statement can be a variable or record component of a reference data type.

The right-hand-side expression must then evaluate to a reference to a record of the same class. The assignment causes the left-hand-side object to reference this same record—no new copy of the record itself is made.

Two reference expressions can be compared for equality by the Boolean relation "=". The relation is true if the two expressions refer to the same record (or both equal null), but not if they refer to different records which are component-wise equal.

Procedure parameters of reference types differ from other parameters in that they are called by reference. They can thus be updated within a procedure call. All other parameters are passed by value.

C. Record creation

There is a special kind of reference expression which can only occur as the right-hand-side of an assignment statement. This is the record-creation expression, and it is the only mechanism by which new records can be created. The record-creation statement causes the object on the left-hand-side to be assigned a reference to the newly-created record.

The form of a record-creation expression is

"record-class-name (sel1:exp1,sel2:exp2,...,seln:expn)"

where each expi is an expression which agrees in type with

the component selected by identifier sel1. All components of the new record must be so initialized, except that reference components may be omitted and initialized by default to null.

Thus, for a record class defined by

```
record class mark(i,j integer; up boolean;
                  end string; f ref(mark))
```

the following statement would create a new record and assign a reference to it to variable x:

```
x := mark(end:'xyz', j:5, up:true, i:0)
```

Then x.f = null, x.end = 'xyz', x.j = 5, etc.

D. Building structures from records

The major importance of records is in their role as building blocks from which data structures are constructed. If record A contains as a component a reference to record B, then records A and B are considered "connected". Then a data structure can be defined as a collection of records which are connected together.

Data structures can be represented pictorially as in Figure 2.1. Each record is drawn as a block which is divided into separate boxes for its component elements. The boxes of one record are labelled with selectors to its components; other records (assuming they are of the same class) by convention follow the same pattern. Each box contains the value corresponding to that component.

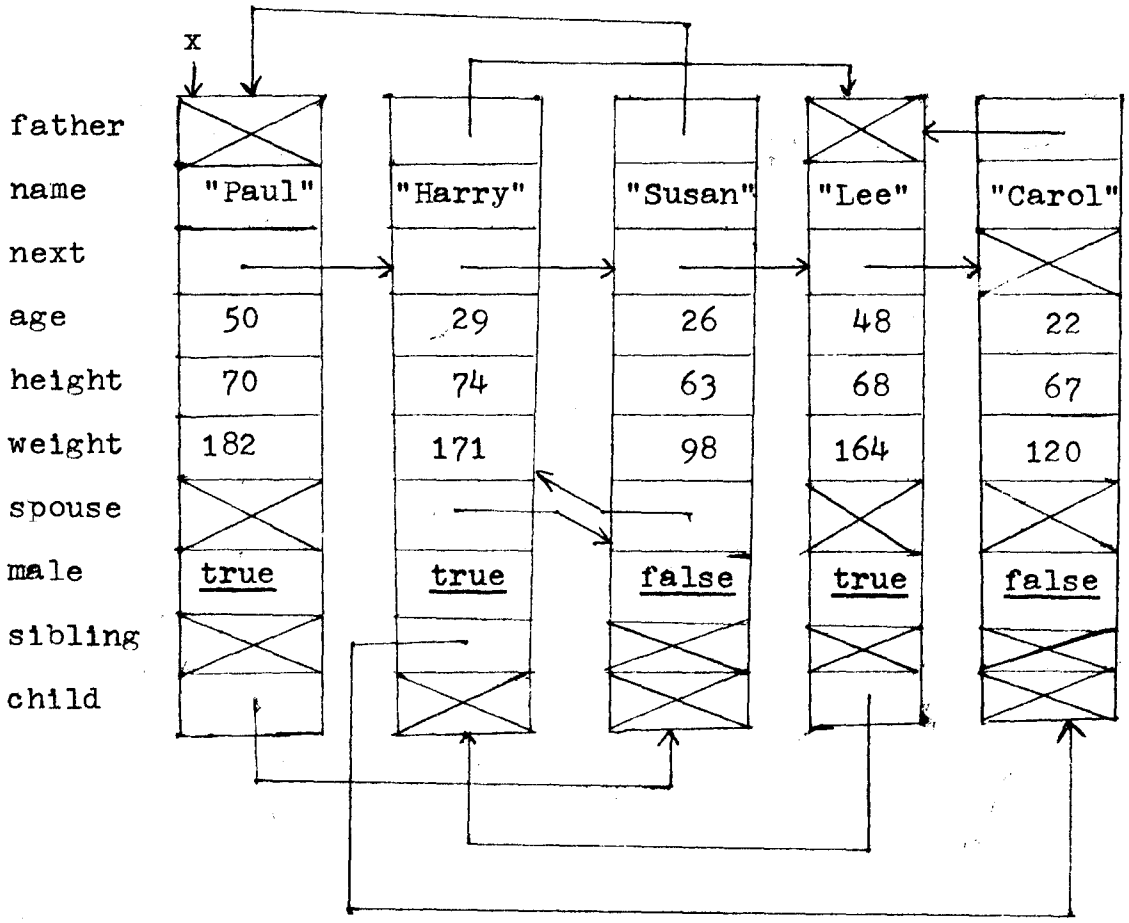


Figure 2.1

Example data structure

A reference value is indicated by an arrow pointing to the referenced record; a cross indicates a value of null.

The records in the figure are all of the following record class:

```
record class person(name string; male boolean;
        age, height, weight integer;
        next, father, child, sibling, spouse ref(person))
```

The data structure pictured illustrates many of the important features of records. The most obvious of these is the usefulness of the record class definition as a template for organizing several groups of data items fitting a common pattern. Each individual record can be used to hold all the information about a separate person.

Storing the information in separate records means that there must be a way to access each record. Rather than requiring a separate variable to reference each record, the records can be linked together into a chain, each one containing a reference to the next. This is the purpose of the "next" component. Variable x references the first record, x.next references the second, x.next.next references the third, etc. until reaching the last record in the chain, whose "next" component equals null. Only one variable is required, no matter how many records are in the structure, and passing of the structure to a procedure, for instance, can be accomplished through this single parameter. In addition, the records can be kept in some fixed order if this is desired.

Reference-typed components are far more versatile than merely a mechanism for chaining records together. They can also be used to represent relations between records. The "father", "child", "sibling", and "spouse" components in Figure 2.1 are employed for this purpose, representing the obvious family relations. A person can have only one father or spouse, so the given component for such a relation simply references that record which is the object of the relation, if it exists, or else equals null.

For a relation like "child", however, where more than one object can exist for a given record, a more complicated technique must be used. The programmer may decide that by convention, the "child" component references the oldest child (i.e. the record corresponding to the oldest child—I will often use the less precise language). The other children are then accessed by following the chain of "sibling" links from one record to another.

An aspect of records which the figure can not illustrate is dynamic allocation. Through the record-creation statement, new records can dynamically be added onto a structure. Similarly, records can be removed from a structure and their storage reclaimed by standard automatic garbage-collection techniques. This facility of dynamic storage management is important in several

applications, such as simulation. The structure in Figure 2.1, for example, could be used to represent a particular family. Changes in the size and structure of the family through time (births, deaths, marriages, etc) could be recorded by dynamically adding and deleting people from the "family tree."

The two features of dynamic allocation and reference-typed components interact in such a way that each requires the other to be really useful. Without reference components, the ability to use dynamic allocation to create new records would be limited by the number of declared reference variables; while an unlimited number of records could still be created, only a fixed number would be accessible at any given time, the rest having been garbage-collected. Without dynamic allocation, on the other hand, it would be impossible to create many structures, and at best awkward to create the rest.

E. Data structuring in other languages

Most modern high-level languages contain facilities for constructing and manipulating data structures. The structural primitives are generally classes of objects analagous to records and references. As noted earlier, the record-handling mechanism described here has been designed to allow a maximum of static type checking.

Additional checking within proofs is required when using a language like ALGOL 68, which allows (the equivalent of) a record class to be the union of distinct classes, or PL/I, where pointers (references) can point to any data object. The major ideas presented in Chapter 3, however, are applicable across a wide range of languages.

Chapter 3

CHARACTERIZING DATA STRUCTURES

In order to prove the correctness of programs which operate on structures, we need the ability to construct assertions which adequately characterize structures. Before this problem can be attacked, we must have a clear idea of what kinds of characterizations are "adequate" for proofs of correctness. If one thinks of a program as a sequence of actions being performed on a group of data objects (obs), then the following dichotomy is helpful: the information contained in an assertion can be divided into those facts which deal with individual obs, and those which relate two or more obs together. The purpose of information of the former kind is to characterize as narrowly as possible the domain of possible values for an ob at the time the given assertion is to be valid. Facts of the latter variety represent joint constraints between the values which can be mutually selected for different obs from their respective domains.

The declarations of a program serve as one means for restricting the domains of its obs. For example, declaring variable i to be of type integer restricts its domain to the integers, or more precisely the subset of the integers representable in the given machine. The domain can be further restricted by a clause in an assertion such as

"$0 \leq i \leq 30$". Similarly, another integer j may be restricted to the domain "$-5 \leq j \leq 5$". A clause in an assertion such as "i = j*j" represents a joint constraint between the values of these two obs. (Note that this particular joint constraint would allow the further restriction of the domain of i to {0,1,4,9,16,25} .)

The distinction between these two kinds of information is actually not as straightforward as the above indicates. The reason is the lack of precision associated with the term "data object". For instance, is an array a single ob, or a whole collection of elementary obs? That is, does the clause

$$(\forall k)(0 \leq k < 100 \supset a[k] \leq a[k+1])$$

restrict the domain of array a, or is it a shorthand way of expressing 100 joint constraints between pairs of obs?

There is no "correct" answer one way or the other. Either interpretation may be the more valid one, depending on how the program operates on the array and its constituent elements. Since there is never any real need to resolve such a point, the ambiguity causes no problems. The reason for raising the issue at all is that it points up the approach taken toward data structures here.

Records play the same role of constituent elements in a data structure as individual array elements do in an array. As with the array, one can imagine situations

in which the individual records should be considered separate obs. The point to be made here, though, is that this thesis is not concerned with such situations. Under those circumstances, records present no new problems in constructing assertions and proofs of correctness; techniques for simple data can be used, with each record component treated as a separate variable. It is only when a group of records is treated as a single structural unit that new tools and methods become needed. In this thesis then, the obs of direct interest will always be data structures, and our concern with records will be limited to the role they play in the structures which include them.

We now have a guideline for organizing our thinking on the issue of characterizing structures. We require two kinds of ability—to restrict the domain of values a structure may assume, and to relate a structure to other obs, both other structures and non-structural data. The problem of restricting structural domains is considered first, since results obtained there influence the approach taken on the other problem.

A. Restricting the domain of a structure

As was noted for elementary obs, the preliminary basis for the restriction of domain of a structure is given by the declarations in the program, in particular

by the record class definitions. For example, consider the record class "person" defined in Chapter 2:

```
record class person(name string; male boolean;
        age, height, weight integer;
        next, father, child, sibling, spouse ref(person))
```

A structure referenced by a variable of type ref(person) is known to belong to a domain P, which can be recursively defined: each element (other than null) consists of a tuple of heterogeneous simple data items—in particular, one character string, one Boolean, and three integers—and references to five (possibly different) elements of P.

The recursion in this definition is automatically introduced by the reference components, and illustrates the inherently recursive form of all structural domains of interest here. The recursion can sometimes be hidden to some extent by using components which reference records of different classes, as in a series of record class definitions like:

```
record class class1 (..., link1 ref(class2));
record class class2 (..., link2 ref(class3));
                .
                .
                .
record class classn (..., linkn ref(class1));
```

The circularity in the pattern of referencing in these n record classes gives rise to n different structural domains, which together form a system of mutually recursive sets.

The only way to define record classes so as to guarantee non-recursive domains is to limit the linking to a fixed finite chain of references. Changing the definitions of one or more of the record classes above in a way which excluded all circular chains of reference components would result in a non-recursive domain. Having the linking completely fixed, however, is tantamount to having no linking at all and employing a single (possibly large) record to hold all the information. Techniques used for elementary data can be employed in such a situation to characterize the data, and therefore cases like this are of no interest here.

Since the structural domains of interest are inherently recursive classes, the natural control structure to use in programs operating on such structures is the iterative loop. Hoare [Hoa72a] has noted the natural correspondence between certain classes of data structures and certain control structures: in particular, recursive data structures correspond to the *while* or *until* loop (with an unbounded number of iterations) just as arrays do to the ALGOL 60 *for* loop, or FORTRAN DO-loop(in which the number of iterations is bounded). The programming of any "interesting" operation on a structure almost always requires at least one loop, driven by iterating a reference variable over the successive records in part or all of a structure.

A1. Patterns

The domain of a structure both before, during, and after such a loop is generally quite a bit more restricted than what is deducible from the record class definitions. All remaining restrictions must be reflected in the corresponding assertions. Because of the loop structure, much of this information takes the form of patterns—relations within and among records which recur regularly throughout the structure or substructure accessed in iterating through the loop. The reason for this is obvious: a single iteration of the loop requires initially and/or causes finally the existence of certain conditions in the local piece of the structure upon which the loop body operates in that one iteration. Since such conditions must prevail over all loop iterations, the result is structure- or substructure-wide patterns.

A key to characterizing the domains of structures, then, is the ability to describe these patterns. The set of records accessed by a loop generally consists of a chain of connected records, so that most patterns encompass a group of records in a chain. We therefore need basic primitives for describing such chains. As the basis of a chain is simply the connection between successive records, the logical primitive concept to consider is "connection."

A2. Direct connection

We think of two records being (directly) connected if a component of one record references the other record. This can be formalized:

> If p and q are reference values, then there is a direct connection from p to q, denoted "p→ q", iff $(\exists \text{sel})(p.\text{sel} = q)$.

Notice that this relation is defined between references to records rather than records themselves. This is necessary since our only access to a record is by an expression which evaluates to a reference to it.

Before proceeding, I should point out and justify a bit of mathematical casualness in this definition. The definition contains the existential quantification of the variable "sel", whose implicit range is over all selectors for the record class of p, i.e. a certain set of character strings. If sel takes on the value 'next', for example, then the value of the expression "p.sel", strictly speaking, is that of "p.'next'", not of "p.next", though the latter is the only interpretation which makes sense in the definition. In a rigorous mathematical sense this is an error, but one which could easily be corrected by the invention of special quoting and dequoting functions. To do so would sacrifice clarity of presentation, however, since these special functions would serve no useful purpose and would simply create

clutter. Since the intent of the above definition is clear, I do not bother to correct the small imprecision. This attitude is carried over later in this thesis when I define and use predicates and functions which include selectors as parameters. It is important to realize that this is simply a matter of convenience, and that the definition as it stands is no less well-defined or precise.

The definition does suffer from a more practical problem, however, in that it is broader than what we generally want. The selector "sel" can be any valid selector for the record class of p, or rather any such selector whose component is of the same type as q. In the case of the record class "person" defined earlier, "p→ q" could mean any of the following:

"p.next = q" or
"p.child = q" or
"p.father = q" or
"p.sibling = q" or
"p.spouse = q".

While in some situations we might wish to express this very fact, we usually want to be more restrictive. In particular, consider the need to describe structural chains, which, after all, provided the direct motivation for defining "connection." The system of linking in a chain is generally limited to a single component in each record, or at most to some subset of possible linking components. We need to be able to limit the connection relation accordingly.

Our means for effecting this restriction is to attach an optional tag to the "$\rightarrow$ " symbol, denoting the component(s) through which the linking is to be limited. To restrict linking to a single component, the selector name is simply written over the arrow:

"x $\xrightarrow{\text{next}}$ y" is equivalent to "x.next = y".

To restrict the linking to several components, a list of their selectors, separated by commas, can be used:

"p $\xrightarrow{\text{a,b,c}}$ q" is equivalent to

"p.a = q or p.b = q or p.c = q".

A more attractive alternative is to let a new identifier denote the set of components, such as s = {a,b,c} , and then simply write "p $\xrightarrow{\text{s}}$ q". The name "s" used for the set, of course, must not be the name of any actual selector.

It should be clear that "p $\xrightarrow{\text{s}}$ q" is more convenient to write than "p.a = q or p.b = q or p.c = q", and its use could be justified on that ground alone, particularly if such a construction occurs numerous times in the assertions of a program. There are at least two other good reasons for this notation, though, which is why we bother with "x $\xrightarrow{\text{next}}$ y" rather than "x.next = y". One is that it gives us a good method for stating several facts like "p.a = q and q.b = r and r.c = t and t.d = u", etc., namely:

$$p \xrightarrow{a} q \xrightarrow{b} r \xrightarrow{c} t \xrightarrow{d} u.$$

The need for this arises particularly when describing chains of links, and the notation here not only handles it concisely, but also affords a graphic picture of the chain. The second reason is that the notation can be directly carried over to the more powerful notion of "ultimate connection."

A3. Ultimate connection

Ultimate connection is the relation existing between two records which are connected by a path of zero or more linking components. Formally, it is the reflexive transitive closure of the direct connection relation, and can be defined:

> If p and q are reference values, then there is an ultimate connection from p to q, denoted "$p \Longrightarrow q$", iff $(p = q)$ or $(\exists r)(p \longrightarrow r \Longrightarrow q)$.

A slightly different concept which is also useful is the non-reflexive transitive closure of "$\longrightarrow$", which indicates a path of one or more links between records:

> "$p \Longrightarrow^{+} q$" iff $(\exists r)(p \longrightarrow r \Longrightarrow q)$.

In both cases, the relations can optionally be tagged so as to restrict the linking to one or more specific components. Note that "$p \xRightarrow{a,b} q$" is equivalent to "$(p = q)$ or $(\exists r)(p \xrightarrow{a,b} r \xRightarrow{a,b} q)$", not to "$p \xRightarrow{a} q$ or $p \xRightarrow{b} q$".

These primitives give us tremendous power to describe chains of records. I indicated previously how to denote

a chain of specific records using a series of "$\rightarrow$" relations. Now we can be more general. If "a" names either the single selector or the set of selectors associated with a chain whose first record is referenced by u, then any record v in the chain is characterized by "$u \overset{a}{\Longrightarrow} v$". Common properties of records in the chain, i.e. patterns, can be described by an expression of the form

$$(\forall v)(u \overset{a}{\Longrightarrow} v \supset \ldots v \ldots)$$

where the right-hand-side of the implication is an expression of the properties for typical record v in the chain.

The best way to illustrate the power of this notation is by an extended example, for which I will use a class of structures like that pictured in Figure 2.1. My belief is that this class of "family trees" is a good one for illustration purposes because virtually any type of structural relation one might wish to represent can be framed in terms of these family relationships. The connection primitives allow us to define any relation we choose. For instance, to say that p is a descendant of z (more precisely, that the person to whom the record referenced by p corresponds is a descendant of the person to whom the record referenced by z corresponds), we can define the following predicate:

$$\text{descendant}(z,p) \equiv (\exists q)(z \xrightarrow{\text{child}} q \xRightarrow{D} p)$$

where $D = \{\text{child}, \text{sibling}\}$.

Similarly, an ancestor predicate can be defined:

$$\text{ancestor}(z,u) \equiv (\exists v)(z \xrightarrow{\text{father}} v \xRightarrow{A} u)$$

where $A = \{\text{father}, \text{spouse}\}$.

More complex relationships like "uncle" can also be handled:

$$\text{uncle}(z,x) \equiv (\exists a,b,c)(z \xrightarrow{\text{father}} a \xRightarrow{\text{spouse}} b \text{ and } (b \xRightarrow{\text{sibling}} + c \xRightarrow{\text{spouse}} x \text{ or } x \xRightarrow{\text{spouse}} c \xRightarrow{\text{sibling}} + b) \text{ and } \text{male}(x)).$$

Figure 3.1 illustrates each of these relations.

As a more general illustration, the whole structural class can be concisely defined. Among the facts characterizing the class are:

(1) All the records are chained together in succession through their "next" components, with the chain ending in a record whose "next" component is null.

(2) Each chain of "sibling"s is sorted by decreasing age.

(3) Every person is in the same "sibling" chain as the (oldest) "child" of his/her "father".

(4) Every person is younger than his/her "father", and older than his/her "child".

(5) Every person who has a spouse is of opposite sex to that spouse, and is the spouse's spouse.

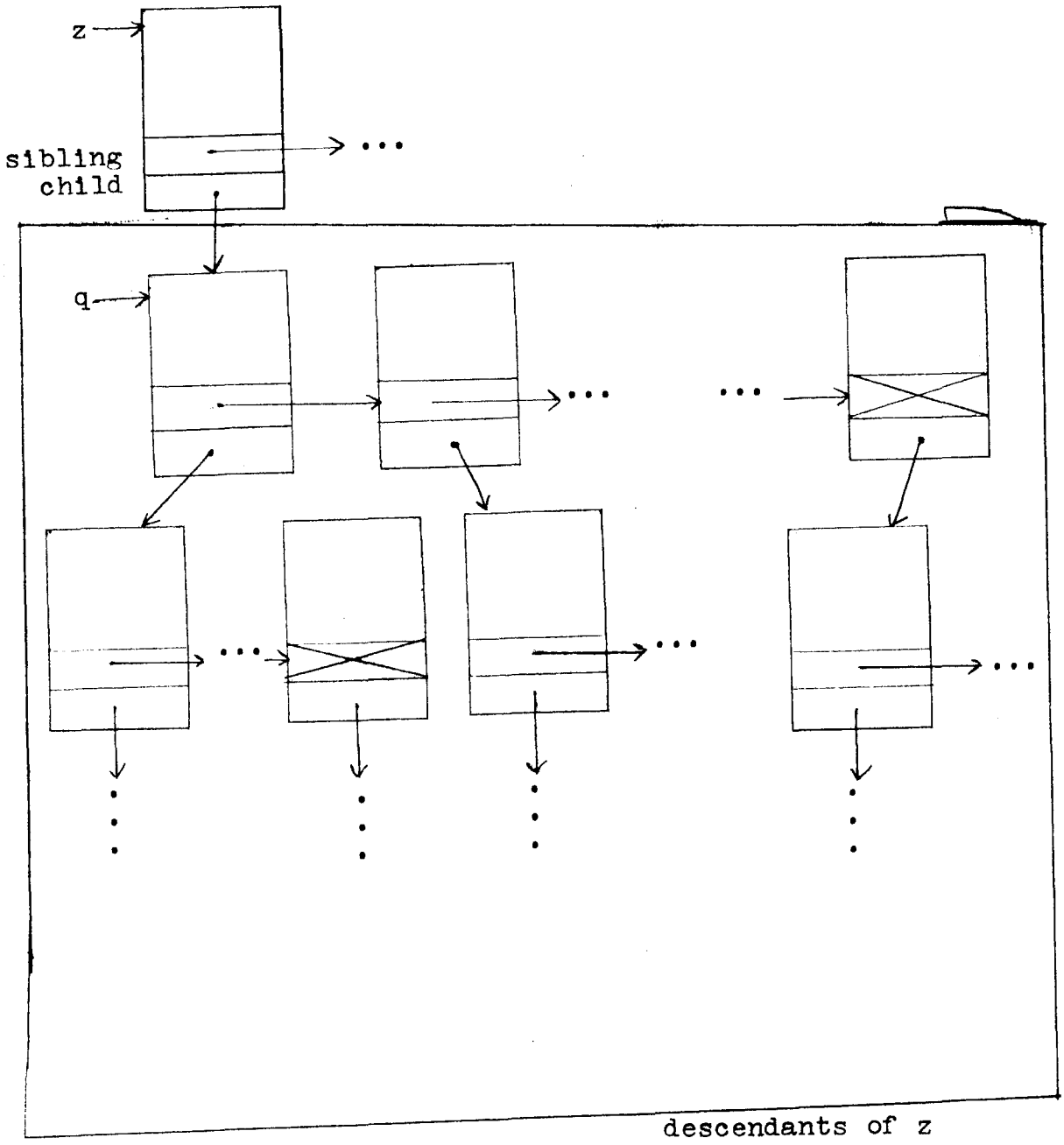


Figure 3.1a
The descendants of z are the records reachable from q = z.child by a path of zero or more "child" and/or "sibling" links.

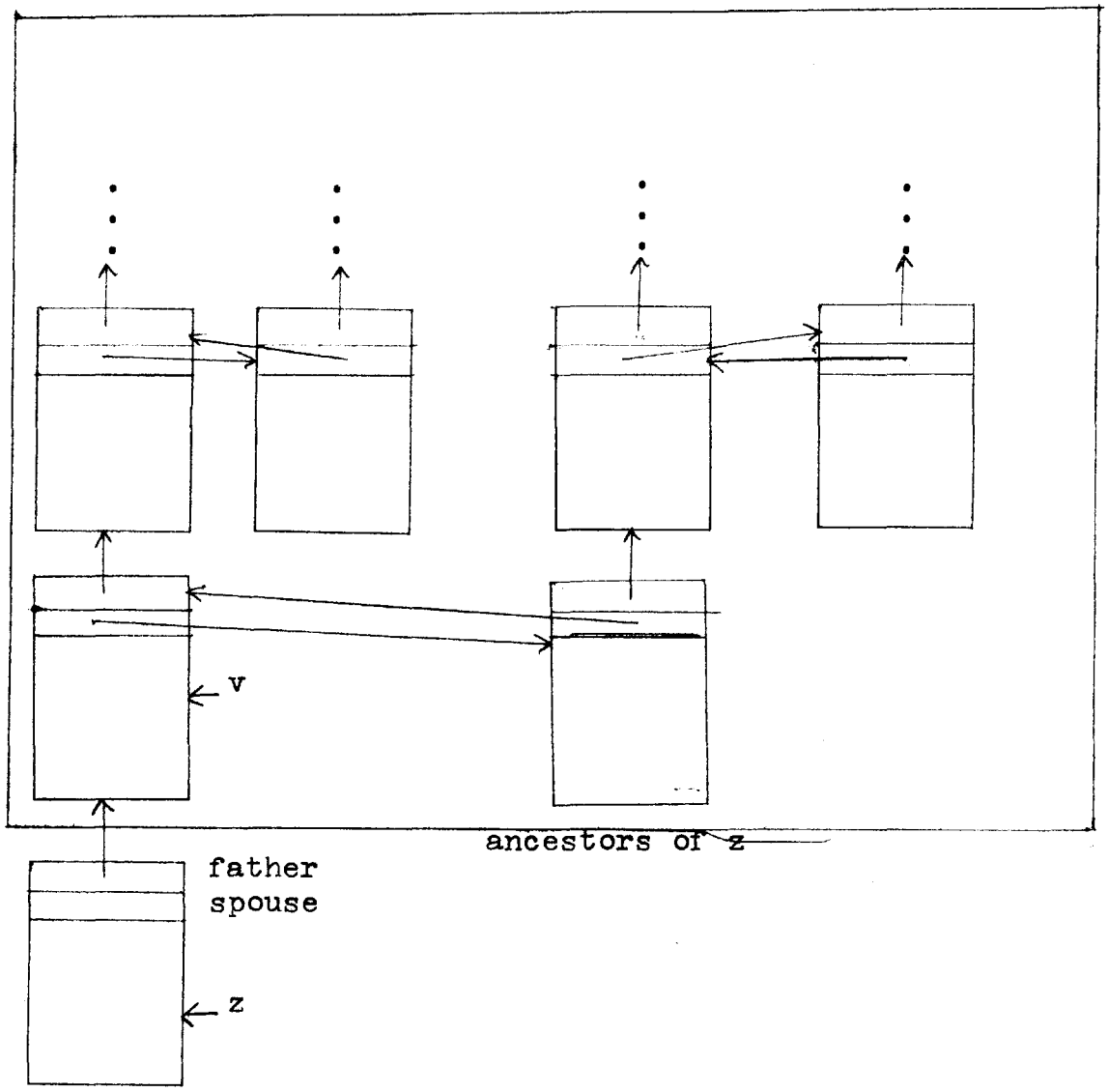


Figure 3.1b
The ancestors of z are the records reachable from v = z.father by a path of zero or more "spouse" and/or "father" links.

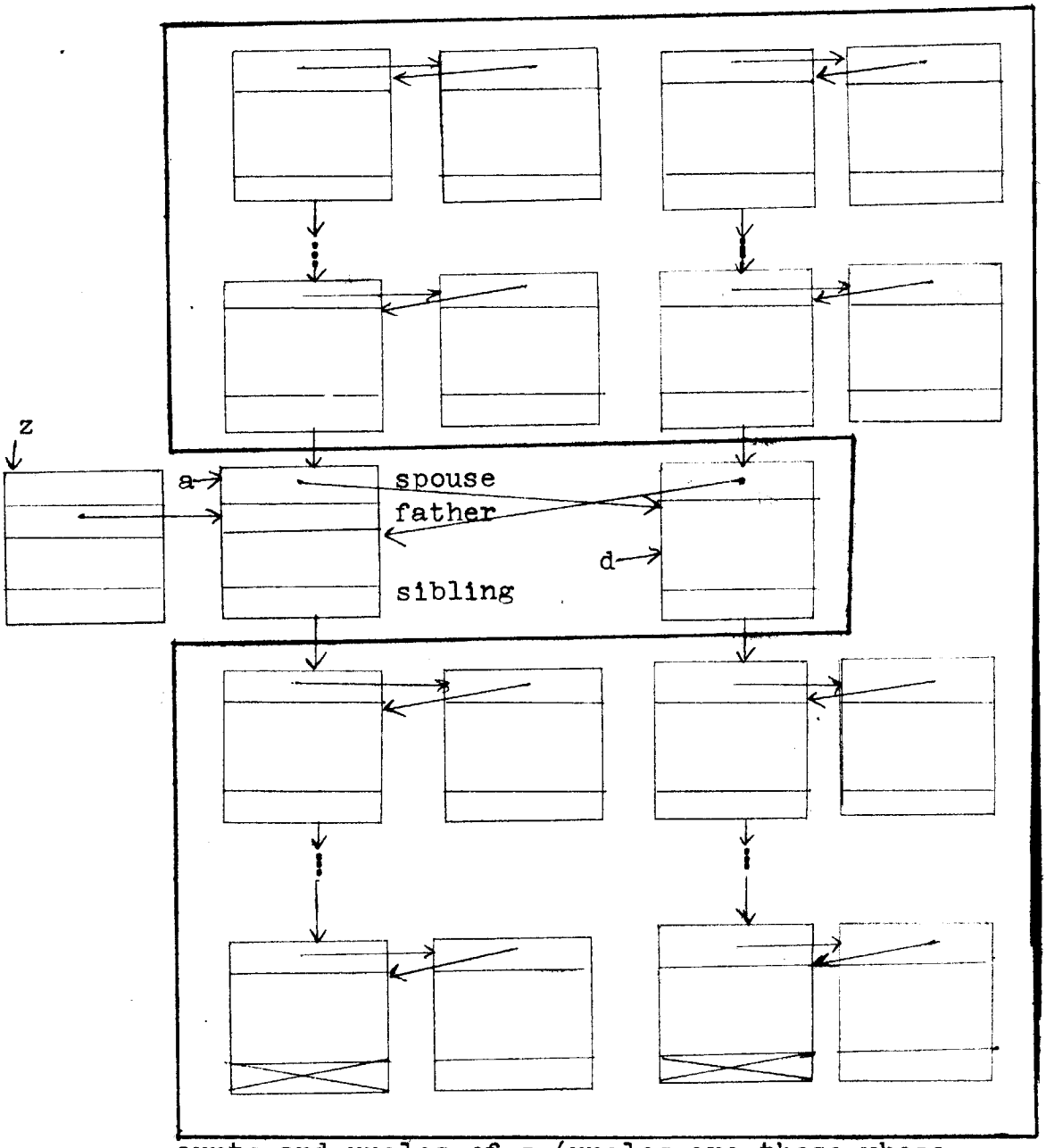


Figure 3.1c
The aunts and uncles of z are the siblings of the parents of z, and their spouses. The parents of z are z.father = a and z.father.spouse = d.

Other details could be added to these, of course, but let us stop here and see how this information can be expressed. Since these properties are all patterns which are in the form of a fact which is true of every record in the structure, they can be stated as applying to a general intermediate record referenced by a universally quantified variable p. If variable x references the first record of the structure, then:

$$
\begin{aligned}
(\forall p,q,r)(x \xrightarrow{next} p \neq \underline{null} \ \underline{and} \ q \neq \underline{null} \ \supset \\
((p \xrightarrow{next} + \underline{null}) \ \underline{and} \\
(p \xrightarrow{sibling} q \ \supset \ p.age \geq q.age) \ \underline{and} \\
(p \xrightarrow{father} q \xrightarrow{child} r \ \supset \ r \xrightarrow{sibling} p) \ \underline{and} \\
(p \xrightarrow{father} q \ \supset \ p.age < q.age) \ \underline{and} \\
(p \xrightarrow{child} q \ \supset \ p.age > q.age) \ \underline{and} \\
(p \xrightarrow{spouse} q \ \supset \ p.male = \underline{not} \ q.male \\
\underline{and} \quad q \xrightarrow{spouse} p)))
\end{aligned}
$$

A4. The invariant

Very often, a programmer will write a number of programs each of which is designed to operate on a structure from the same structural class. It may be that certain of these programs are "top-level" routines and can each call certain others as subroutines, which in turn can each call others, etc. Or, the programs may all be independent, each one performing a specific

basic operation on the structure, such as addition or deletion of a record, tests for certain predicates, rearrangement, etc. In any case, the structure possesses a lifetime which transcends the scope of one single program.

In this situation, the restriction of the structure's domain to the given class assumes an importance far greater than when such a restriction is limited to a single point in time (i.e. a single assertion). The restriction now becomes a fundamental aspect of the structure, expressing what Foley and Hoare[Fol71] call the "purpose" of the structure. Following their terminology, the expression of this restriction is referred to as the invariant of the structure. In the section just previous, for instance, the invariant of a "family tree" was constructed.

The invariant of a structure plays an important role in inductive assertion proofs of programs operating on the structure. Any program which takes the structure as an input should include the invariant within its input assertion. Every program which outputs the structure must include the invariant within its output assertion. It will also be the case that most, though not necessarily all, of the intermediate assertions of programs which operate on the structure will include the invariant.

Not all the assertions must include the invariant because it need not be truly "invariant" on a statement-to-statement basis, but only a program-to-program basis. In fact, the validity of the invariant will often lapse in the middle of a computation which performs a reasonably complex operation on the structure, or even such basic operations as insertion and deletion of records. Sections of code in which the invariant is temporarily invalidated can be considered critical sections. In the parlance of multiprogramming and concurrency of active procedures, a piece of code is called a "critical section" if its action must not be interrupted by that of another active process. An example would be the code between the addition of an item onto a stack and the subsequent updating of the stack pointer. Such a critical section is usually protected from interruption by use of a semaphore.

Even though the programs under consideration are purely sequential, with no parallel concurrent activity, the concept of a "critical section" is still useful. Since the invariant expresses the "purpose" of the structure, and since other programs operating on the structure rely on the validity of the invariant, it is critical that any program which invalidates the invariant must reestablish its validity. In addition, it is the critical section which often performs the major work of the program.

The act of proving the program correct will, of course, verify the action of restoring the invariant. This is guaranteed by the necessary inclusion of the invariant in the output assertion.

Use of the invariant offers a number of advantages in assertion proofs. Syntactically, it allows assertions to be more concise: the invariant need only be stated in its entirety once, at which point a convenient predicate (possibly in terms of one or more parameters) can be defined to represent it. This predicate can then be used in any assertions which include the invariant. This helps prevent the assertions from being filled with a lot of repeated, unchanging details. It also serves to separate the information within an assertion into two kinds: that which changes in the course of a program, and that which does not. This makes assertions easier to read and follow. Stating the invariant at the beginning of the program also allows the reader to see at a glance what kind of structure the program is designed to operate upon.

The greatest advantage of the invariant is in the proof of correctness itself. In the programs which operate on the structure, most if not all assertions will contain the invariant, so that most if not all verification conditions will require a proof of the continuing

validity of the invariant. There are only a limited set of ways to operate on a structure in a small, loop-free section of code (i.e. the body of a v.c.); hence, it is quite likely that a proof of correctness will require numerous applications of a result like: "A statement of the form 'p.link := q.link' does not invalidate the invariant provided that...." It makes sense then to simply prove the result once as a lemma, after which the lemma can be invoked in any proof which requires the result.

More generally, one can analyze all statements which might alter the structure, and prove a whole series of lemmas to decide in all cases whether the invariant is maintained. This may seem like a massive undertaking, but in fact it is not, as illustrated in Chapter 4. Proving the lemmas before attempting the actual proofs of correctness vastly simplifies the verification process. In fact, it may be helpful to state the lemmas before even writing any programs to operate on the structural class; the insights gained in the former task may prove beneficial for the latter.

A5. Program invariants

In proving the correctness of a program operating on structures from a given class, the invariant for that class will form part of each assertion in the program.

It may also be the case that over the course of the one program, certain other facts about the structure remain constantly true as well—the domain of the structure within that program may be restricted to some subset of the larger class defined by the invariant. It probably does not pay to define a new class invariant and construct the attendant series of lemmas for use in proving the one program correct. However, it would be nice to obtain the syntactic benefits of the invariant, the conciseness and structure afforded to the assertions.

This can be accomplished by use of a program invariant, which plays a role similar to that of the class invariant, but on a scope local to a single program. The program invariant includes the class invariant and may also include any number of other clauses. It is stated at the very beginning of a program (in a comment) and an abbreviation is generally given there to represent it in assertions. Although this abbreviation only takes one clause of an assertion syntactically, each actual clause of the program invariant must be proved true in each v.c.

This concludes the discussion of how to restrict the domain of a structure. To summarize the results: the connection primitives give us a means for describing intra-structural relations; using them in conjunction

with techniques developed for simpler data and the constructs of predicate calculus such as quantification, we can define concisely any structural class, and incorporate the definition into a class invariant. Lemmas can then be proved about the effects of various source-language statements on the validity of the invariant. This allows the characterization of the structural class to be integrated into the actual proof of correctness. These concepts are put to use in proving actual programs in Chapter 4. To construct such proofs, however, we must address the other major issue mentioned at the beginning of this chapter—relating structures to other data.

B. Relating a structure to other data

To construct proofs of correctness involving a data structure, one must be able to express relations between the structure and any other ob a program can contain. This includes both elementary variables and also other structures. Perhaps most importantly, the value of a structure at a given point in time must be related to its value at a different point, such as its initial value; for any program manipulating the structure, we need to relate the final output structure to the initial input structure, in order to express the intended result of the program in the output assertion. Since the program invariant states what is unchanged about the structure,

what is required additionally is the ability to precisely describe changes in the structure due to the action of the program.

B1. Containment relations

The relation of a data structure to another ob is often one of containment. For instance, we may wish to express the fact that the structure (or some substructure of it) contains a record with a certain component equal in value to an elementary variable. Or, that a smaller structure is isomorphic (equal in all non-reference components, corresponding in its linking) to some sub-structure of the larger structure. For a large class of relations such as these, the basic issue involved is containment. To express such a relation, we need to be able to refer to internal information within a data structure. The intra-structural primitives for connection relations, along with more basic tools used for simpler data, provide the mechanism for doing so.

For example, to state that a Boolean variable "in" expresses whether or not integer "i" is equal to the "num" component of any record in a chain linked by "next" components, starting from record "x":

$$in = (\exists p)(x \xrightarrow{next} p \neq \underline{null} \ \underline{and} \ p.num = i).$$

Expressing the substructure-isomorphism relation is slightly more complicated. Since the concept of

isomorphism is recursive, an auxiliary recursive definition is employed:

$$(\exists\, z)(x \Longrightarrow z \quad \underline{\text{and}} \quad \text{isomorphic}\,(z,p))$$

$$\underline{\text{where}} \quad \text{isomorphic}(\underline{\text{null}}, \underline{\text{null}})$$

$$\underline{\text{and}} \quad \text{isomorphic}(q,r) \equiv$$

$$(\forall\, \text{sel} \in \text{Nonref})(q.\text{sel} = r.\text{sel}) \quad \underline{\text{and}}$$

$$(\forall\, \text{link} \in \text{Ref})(\text{isomorphic}(q.\text{link}, r.\text{link})).$$

Nonref and Ref designate the sets of all non-reference selectors and reference selectors repsectively for the given record class(es). Notice that this definition works only for structures in which cycles and shared substructures are not permitted. It could justifiably be used, then, only if the program invariant for the structure ensured this condition.

This illustrates an important point. Our separate considerations of the two issues of domain-restriction and relations-to-data should not mislead one into supposing that there is no interaction between these two areas. In particular, the restriction of a structure to a specific domain can often be utilized to simplify the statements of its relations to other obs. A major use of this fact will be made shortly.

B2. Abstract relations

Unfortunately, there are many situations where one needs to express relations other than those in the broad

category of "containment." This often includes the very important case of relating the values of a structure on input to and output from a program. Consider, for example, the very simple operations of adding a record to and deleting a record from a structure. If the addition or deletion takes place somewhere in the middle of the structure, then there is no substructure-like relationship between the initial and final values of the structure. It is likely that one would not know at all where the given record actually is: the addition may be constrained so as to maintain some aspect of the invariant (such as an ordering relationship); the deletion might be based on the matching of some component to a key value. An accurate expression of the change in terms of the internal connections between records may be impossible, or else prohibitively complex. Yet these operations appear quite straightforward, and can be described in English rather simply. Why then the trouble?

The reason for the difficulty is that the computational manipulation being performed is indeed complex. (If it were not, then the program to accomplish it, and hence its proof of correctness, would be trivial.) The surface simplicity is due to our ability to think and speak in terms of abstract objects and operations. Record addition

and deletion correspond in the abstract to adjoining and removing an element from a set, respectively. Given a way of mapping the structure into the set in question, the results can be easily stated in terms of simple set operations:

Addition: (Set mapped into by final structure) =
(Set mapped into by initial structure) U
{(given element)}

Deletion: (Set mapped into by final structure) =
(Set mapped into by initial structure) -
{(given element)}

Addition and deletion are among the simplest operations for which such a viewpoint can be taken. Numerous program manipulations which are complex computationally become considerably simpler when viewed as their correponding abstract operations. This suggests an alternative approach for relating structures to program data: map structures into abstract data objects (such as sets) and then express the relations in the abstract domain. To do this, though, we must have a way of formalizing the mapping.

B3. Representation functions

A vehicle for formalization is the representation function, a concept proposed by Hoare [Hoa72b]. A representation function maps elements from a given class of computational objects into a set of abstract data objects. Such a function can be defined in any way, so

long as its domain and range are appropriate to the above statement. Because of the generally recursive nature of structures, recursive function definitions are common.

A class of computational objects can be held in correspondence simultaneously with several abstract data classes. There is thus no single "correct" representation function for a given structural class, but rather several functions which can be used. One or another of these functions may prove especially helpful for describing the operation of a particular program, and hence be used in its assertions. The sole criterion for choosing a particular representation function is the extent to which its use contributes to the clarity of assertions and the ease of proof for the given program.

While representation functions help to clarify the meaning and reduce the size of assertions, their use is not without cost. In the actual proof of a program's correctness, representation functions are a new class of alien objects which must be dealt with. The problem is not so much with the abstract data objects returned by the functions. Use of representation functions returning sets, for instance, may mean that the proof of correctness requires certain results of set theory. The results needed, however, are unlikely to be terribly

difficult or profound. The only possible problem might arise in connection with eventual mechanization of proofs—set theory is that much more mathematical material which must be covered by a mechanical program verifier's theorem base.

The more serious aspect of the problem concerns the representation functions themselves. By including within assertions expressions which involve representation functions, we force the proofs of the v.c.'s to prove results about the values of these functions. In particular, we must prove how each statement in a program affects the value of each representation function included in its assertions.

This is a familiar problem, for it corresponds exactly to one we faced earlier with regard to class invariants. A similar solution can be adopted: analyze all cases of interest beforehand, and prove lemmas on the effect of each kind of program statement on the value of the representation function. Using the representation function then actually simplifies the proof of correctness, since much of the unimportant detail of a proof can be submerged in the invoked lemmas.

This would not be a practical solution if one defined a new special-purpose representation function for each structure in each program. It would just not be worth-

while to bother proving the whole set of lemmas if they were to be used for proving only the one program. By restricting the domain of each representation function to a specific structural class (i.e. by associating the representation function directly with the class invariant), though, general-purpose representation functions can be defined which are applicable to all structures of the given class.

The number of representation functions required for a given structural class need not be very large. The fact is that there aren't very many kinds of abstract data objects, as Hoare[Hoa72a], for example, shows. In fact, one really needs only two broad classes—sets and sequences—corresponding to whether the given collection of constituent elements is considered to be unordered or ordered, respectively. Other abstract data classes which are potentially useful all turn out to be special cases of these two; e.g., stacks and queues are two special cases of sequences in which only certain operations are permitted. Most structural classes therefore require representation functions yielding sets and sequences only, and usually this means just one function for each of the two. This is because parameters of the functions can be used to determine which links to follow in the chain and which (component) values are to be used as set or sequence elements.

C. Summary

To summarize my approach, the restricted structural domain is used as the basis for constructing assertions about and proving programs operating on data structures. Such a domain can be defined by describing intra-structural relations using the primitives of direct and ultimate connection, and combining these relations with standard techniques to specify the class restrictions. This definition of the class is embodied in the class invariant. Structures in the class can be mapped into corresponding abstract data objects by defining representation functions on the domain which yield sets and sequences. These abstract data objects can then be used to express relations between the structures and other data in the program. Inclusion of the invariant and the representation functions in assertions leads to the necessity of proving results about them in the course of proving the correctness of programs. This task is simplified by analyzing the various kinds of source-language statements, and proving lemmas concerning the effect of each on the invariant and the values of the representation functions. Chapter 4 illustrates the application of this approach on a particular structural domain, and demonstrates the proofs of correctness which result.

Chapter 4

SINGLY-LINKED LISTS

The basic outline of my approach to handling data structures in verification proofs was presented in Chapter 3. Several new concepts were introduced, explained, and illustrated by examples. However, in order to really understand all the concepts, both individually and in relation to each other, one must follow through at least one complete application of the approach—from the choice of a particular class of structures to one or more actual proofs of correctness. This chapter takes a structural class which is particularly simple, and thus good for illustration purposes, and applies the approach, ultimately proving three programs correct which operate on structures of the class.

Consider the following simple technique for constructing a data structure: take some number of records all of the same class; make sure that the class is defined so that at least one record component is itself a reference to records of the class; choose one reference component as the linking component for the structure; put the records in some type of order, based on some characteristic or completely arbitrary; connect the records together by having the link in each record reference the next record in succession, with the last record's link component equaling null.

The resulting structure, which will look like the one in Figure 4.1, is a singly-linked list, henceforth simply called a "list."[1] Because of their simplicity and wide versatility of application, lists are one of the most widely-used classes of structures. These qualities also make them attractive for use here. Because of their versatility, a wide range of program applications are available for use in illustrating the concepts presented. Their simplicity of form makes them easy to use, and means that there will be a minimum of extraneous details to distract attention from the important concepts.

A. Definition

The first item of business is to define the class of lists formally. The easiest way to do this is by a recursive definition. If the record class to which the records in the list belong is named "rc" and the link component selector is "next", then:

Definition L.1 A list is referenced by either null or by reference to a record of class "rc" whose "next" component references a list.

This definition is theoretically sufficient, since it provides an effective test procedure for determining membership in the structural class. The main use for a structural class definition, however, is in the invariant

[1] Notice that this is not a LISP-type "list", for which I will use the alternate term "List", after Knuth[Knu68].

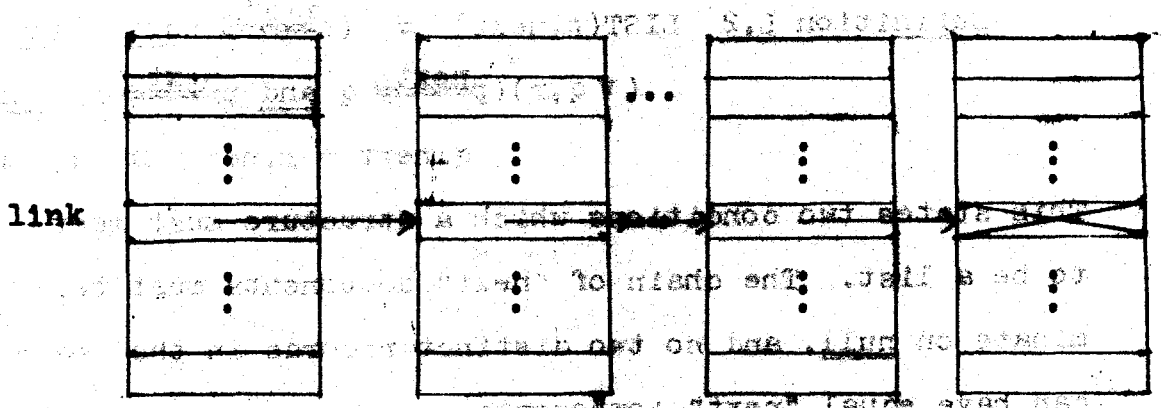


Figure 4.1 A typical list

for a structure. For this purpose, what is needed is what I call an analytic definition—one which analyzes the structural properties which are characteristic of the class. This is provided by the following predicate, which takes as parameters a reference to the structure and the selector for the link component:

Definition L.2 LIST(p,next) $\equiv$ ($p \xRightarrow{next}$ null and
($\forall$ q,r)($p \xRightarrow{next}$ q and $p \xRightarrow{next}$ r and
q.next = r.next $\supset$ q = r))

This states two conditions which a structure must meet to be a list. The chain of "next" components must terminate on null, and no two distinct records in the chain can have equal "next" components.

The two definitions for lists are in fact equivalent:

Theorem L.3 LIST(p,next) iff
p references a list with "next" as the link component according to Definition L.1.

The proof of this theorem is given in Figure 4.2.

The following corollaries to these definitions are immediately apparent:

Corollary L.4 LIST(p,next) and ($p \xRightarrow{next}$ q) $\supset$
LIST(q,next).

That is, any record within a list is itself the beginning of a list. This is fairly obvious from Definition L.1.

Theorem L.3 LIST(p,next) iff

p references a list with "next" as its link component according to Definition L.1.

Proof: by induction on n, the number of records in the structure referenced by p.

Basis: n = 0: p references a structure with 0 records iff p = null. null references a list according to the first definition, and "LIST(null,next)" is true according to the second. So the two definitions are equivalent for the basis case.

Induction step: Assume the theorem is true for structures of size (n - 1), where $n \geq 1$. Prove that it is satisfied for structures of size n:

If p references a list according to Definition L.1, then the first record's "next" component references a list. By the induction hypothesis, this list also satisfies Definition L.2. Therefore,

$$p.next \overset{next}{\Longrightarrow} null \quad \text{and}$$

$$(\forall q,r)(p.next \overset{next}{\Longrightarrow} q \;\&\; p.next \overset{next}{\Longrightarrow} r \;\&\; q.next = r.next \supset q = r).$$

So, "$(p \overset{next}{\Longrightarrow} null)$" follows immediately from the definition of "$\Longrightarrow$".

Figure 4.2

Proof of Theorem L.3

For any q and r such that "$p \xrightarrow{next} q$ and $p \xrightarrow{next} r$ and q.next = r.next" is true, either "$p.next \xrightarrow{next} q$ and $p.next \xrightarrow{next} r$" is true, or at least one and possibly both of q and r are equal to p, by the definition of "$\Rightarrow$". If the first possibility, then q = r from above, the induction hypothesis. If both q and r are equal to p, then clearly q = r.

If one but not both are equal to p, then without loss of generality, let it be q. Then q = p, and $p \xrightarrow{next}{}^{+} r$. So $p.next \xrightarrow{next} r$, and so $p.next \xrightarrow{next}{}^{+} r.next$, each of these last two following by the definition of "$\Rightarrow$". But since r.next = q.next, and q = p so that q.next = p.next, this means that $p.next \xrightarrow{next}{}^{+} p.next$, and it can easily be shown that this contradicts the induction hypothesis that $p.next \xrightarrow{next} \underline{null}$. (Basically, the argument runs that the structure must be circular, and thus cannot terminate on null.) So this case is impossible, and both other cases confirm that the second clause of Definition L.2 is satisfied. Since it was shown above that the first clause is also satisfied, this proves that Definition L.1 implies Definition L.2, which completes half the proof.

Figure 4.2(page 2)

If "LIST(p,next)" is valid according to Definition L.2, then $p \overset{next}{\Longrightarrow} \underline{null}$ and
$(\forall q,r)(p \overset{next}{\Longrightarrow} q \ \underline{and} \ p \overset{next}{\Longrightarrow} r \ \underline{and} \ q.next = r.next \supset q = r)$.
Since $n \geq 1$, "p.next" is well-defined, and in fact
"$p.next \overset{next}{\Longrightarrow} \underline{null}$" is true by definition of "$\Longrightarrow$" and the fact that $p \overset{next}{\Longrightarrow} \underline{null}$ above; similarly,
"$(\forall q,r)(p.next \overset{next}{\Longrightarrow} q \ \underline{and} \ p.next \overset{next}{\Longrightarrow} r \ \underline{and} \ q.next = r.next \supset q = r)$"
follows <u>a fortiori</u> from the similar clause above for p. So, by the induction hypothesis, p.next references a list according to Definition L.1, and by that definition, this means that p references a list. So Definition L.2 implies Definition L.1.

Since the validity of each definition implies that of the other, they are indeed equivalent for n-record structures, and by induction, for all structures.

Q. E. D.

Figure 4.2(page 3)

Corollary L.5 LIST(p,next) $\supset$
($p \xRightarrow{next} q$ and $q \xRightarrow{next} p$ iff $p = q$).

So, no two distinct records can each be within the list headed by the other.

B. Representation functions

The next item of business is to define general-purpose representation functions on lists. We require mappings from lists to both sets and sequences. The mappings defined here are quite simple—into the set or sequence of successive values of one particular component from each record in the list in turn. These functions are adequate for a wide range of situations, including the three example proofs presented later in the chapter. A more generalized version of the functions would map a list into a set/sequence whose elements are each a tuple of component values from a single record. For the sake of simplicity, we content ourselves with the simpler functions, but it should be observed that the treatment required for the generalized functions would be directly analagous to that presented here for the simpler mappings.

Since the particular component selected as set or sequence element will vary from list to list, and even from one application to another, it makes sense to include the selector to this component as a parameter to the representation functions. Other parameters which

are needed are a reference to the beginning of the list, and the link component selector, in order to get from one record to the next. With these parameters, we can now take advantage of the recursive form of the list structure to define the following two functions:[1]

Definition L.6 LISTSET(p,sel,next) ≡

if p = null then ∅

else {p.sel} U LISTSET(p.next,sel,next).

Definition L.7 LISTSEQ(p,sel,next) ≡

if p = null then [ ]

else concat([p.sel],LISTSEQ(p.next,sel,next)).

These functions perform the obvious task of accessing the selected component of each record in the list in succession, and adding it to the set or concatenating it onto the sequence, respectively.

Two other functions are useful enough to define here. They represent the analogs to the above two functions for sublists, rather than full lists. The bounds of the sublist are indicated by two parameters, one of which references the first record in the sublist, and the other of which references the record immediately following the

[1]The notation used for sequences here is that "[ ]" represents the empty sequence, "[a]" represents the one-element sequence with single element a, and "concat" is the operation of concatenation on sequences.

last record in the sublist.[1] The two functions are defined in the same way that the two functions for full lists were:

Definition I.8 SUBLISTSET(p,q,sel,next) ≡

if p = null or p = q then ∅

else {p.sel} U SUBLISTSET(p.next,q,sel,next).

Definition I.9 SUBLISTSEQ(p,q,sel,next) ≡

if p = null or p = q then []

else concat([p.sel],SUBLISTSEQ(p.next,q,sel,next)).

The test "p = null" is required in these two definitions to ensure that the functions be well-defined even when q does not reference a record in the list, i.e. when "$p \xrightarrow{next} q$" is false. Neither of these two functions, nor the two for full lists, are guaranteed to be well-defined, however, unless LIST(p,next) .

Finally, notice that a list is merely a sublist which terminates on null, so that:

Corollary I.10 SUBLISTSET(p,null,sel,next) =

LISTSET(p,sel,next).

Corollary I.11 SUBLISTSEQ(p,null,sel,next) =

LISTSEQ(p,sel,next).

---

[1] The reason for this seemingly strange convention is that it allows for more general utility. If one desires to represent a sublist whose final record is referenced by "x", then this can be accomplished by using "x.next" as the second parameter to these functions. Were the opposite convention used, it would be difficult, if not impossible, to express the non-inclusive case, due to the absence of backward referencing in lists.

There are numerous other corollaries to the above definitions. Some of the more useful ones are:

Corollary L.12

LIST(p,next) and ($p \overset{next}{\Longrightarrow} q \overset{next}{\Longrightarrow} r$) ⊃

(SUBLISTSET(p,r,sel,next) =

SUBLISTSET(p,q,sel,next) U SUBLISTSET(q,r,sel,next)

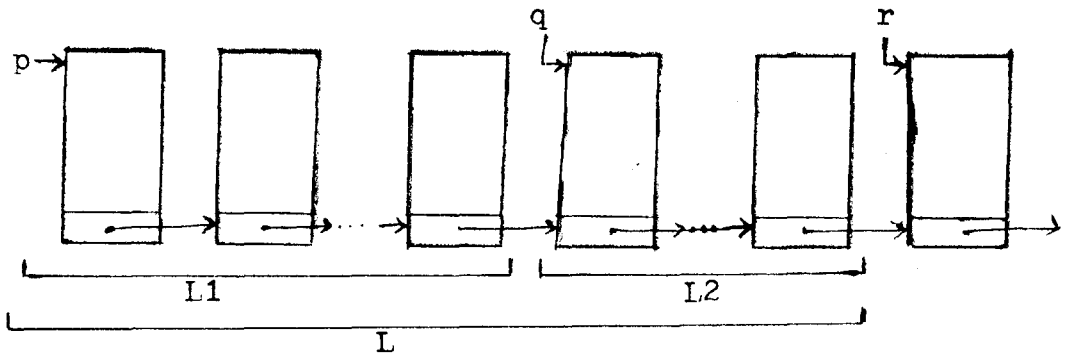
and

SUBLISTSEQ(p,r,sel,next) =

concat(SUBLISTSEQ(p,q,sel,next),

SUBLISTSEQ(q,r,sel,next))).

This corollary deals with a situation like that illustrated below:



That is, sublist L is the composition of sublists L1 and L2. In that case, the set mapped into by L is the union of the sets mapped into by L1 and L2, and the sequence mapped into by L is the concatenation of the sequences mapped into by L1 and L2. Notice that if r = null, this result can be used for lists L and L2 rather than sublists, using Corollaries L.10 and L.11.

Corollary L.13 LIST(p,next) and p ≠ null ⊃

(SUBLISTSET(p,p.next,sel,next) = {p.sel} and

SUBLISTSEQ(p,p.next,sel,next) = ⟨p.sel⟩).

This simply gives the values of the representation functions for the special case of one-element sublists.

Corollary L.14 LIST(p,next) and not($p \xRightarrow{next} q$) ⊃

(SUBLISTSET(p,q,sel,next) = LISTSET(p,sel,next)

and

SUBLISTSEQ(p,q,sel,next) = LISTSEQ(p,sel,next)).

This result is for the case that the second parameter to the functions for sublists, q, does not reference a record in the list referenced by p. The values of the representation functions are then the same as if the second parameter were null.

Corollary L.15 LIST(p,next) and

($p \xRightarrow{next} q \xRightarrow{next} + r$) ⊃

q.sel ∈ SUBLISTSET(p,r,sel,next).

That is, for any record q within a sublist, the component q.sel is guaranteed to belong to the set represented by the sublist.

Corollary L.16[1] LIST(p,next) and ($p \xRightarrow{next} + r$) ⊃

(p.sel = first(SUBLISTSEQ(p,r,sel,next) and

SUBLISTSEQ(p.next,r,sel,next) =

final(SUBLISTSEQ(p,r,sel,next))).

---

[1]See Appendix A for definition of the primitive operations on sequences—first, final, initial, and last.

The first element of the sequence is the component of the first record, and the rest of the sequence is mapped into by the rest of the list or sublist.

Corollary L.17[1]

$$\text{LIST}(p,\text{next}) \text{ and } (p \xrightarrow{\text{next}}{+} r \neq \text{null}) \supset$$
$$(r.\text{sel} = \text{last}(\text{SUBLISTSEQ}(p,r.\text{next},\text{sel},\text{next}) \text{ and}$$
$$\text{SUBLISTSEQ}(p,r,\text{sel},\text{next}) =$$
$$\text{initial}(\text{SUBLISTSEQ}(p,r.\text{next},\text{sel},\text{next}))).$$

Similarly, the last sequence element is the component of the last record, and the initial subsequence is represented by the initial sublist.

C. Verification lemmas

We have now reached the stage where we can construct a series of lemmas to formalize the effects of various source-language statements on the LIST invariant and on the values of the representation functions defined on the class. One might suppose that this is going to be a monumental task, but in fact it is surprisingly easy. For a number of reasons we are able to handle all relevant cases in only a handful of lemmas.

One reason is that all our results are framed in the most general terms possible. That is, even though the records in a list which are most frequently operated upon are the first and last, it is not necessary to consider

[1]See the footnote on the previous page.

these special cases separately. The recursive nature of lists makes all results on intermediate records uniformly applicable. We also handle all effects of a given program statement in one lemma—we do not bother with separate lemmas stating the effect on the invariant and each representation function, but instead lump all these results together in one lemma.

Finally, we take advantage of Corollaries L.10 and L.11, the fact that LISTSET and LISTSEQ are only special cases of SUBLISTSET and SUBLISTSEQ. Results need only be expressed in terms of the latter two functions.

The lemmas stated here are presented without proof. The reason is that the proofs both lack any instructive value and are rather tedious. To give the reader the flavor of such proofs, Appendix B contains the proof of one of the lemmas, Lemma L.19. The proof uses the verification axioms of Hoare [Hoa69] as its logical foundation, and is illustrative of the proofs required for the other lemmas.

While the results embodied in the lemmas are complicated to state, they are all really completely straightforward, following directly from the definitions of the invariant and the representation functions. Accompanying each lemma is a brief discussion explaining the result. All lemmas employ the convention that free variables are

assumed to be universally quantified.

The nomenclature for the statement of the lemmas is borrowed from Hoare [Hoa69] and was introduced earlier. The basic form is "[P] {Q} [R]", where Q is the code (in this case, statement) for which the lemma applies, P is the necessary precondition for application of the lemma, and R is the resultant postcondition. An "i" subscript refers to values initial to the execution of the code. In proving programs later in this chapter, "i" subscripts will similarly be used to denote values which are initial to the verification condition under consideration. An "o" subscript is reserved to denote values initial to the entire program. Only the "o" subscript can be used in assertions.

The simplest operation possible on a list is to update a non-link component in one of its element records. The result is to leave the invariant valid, the representation functions for the given component changed by the single substitution, and representation functions for all other components unchanged. This is expressed:

Lemma L.18

$$\left[\text{LIST}(p,\text{next}) \ \underline{\text{and}} \ (p \overset{\text{next}}{\Longrightarrow} q \overset{\text{next}}{\Longrightarrow} + r)\right]$$

$$\{q.\text{sel} := \text{expr}\}$$

$$\left[\text{LIST}(p,\text{next}) \ \underline{\text{and}} \ (p \overset{\text{next}}{\Longrightarrow} q \overset{\text{next}}{\Longrightarrow} + r) \ \underline{\text{and}}\right.$$

$$\text{SUBLISTSET}(p,r,\text{sel},\text{next}) =$$

$$(\text{SUBLISTSET}(p,q,\text{sel},\text{next}))_i \ \cup \ \{\text{expr}\}$$

$$\cup \ (\text{SUBLISTSET}(q.\text{next},r,\text{sel},\text{next}))_i \ \underline{\text{and}}$$

$$\text{SUBLISTSEQ}(p,r,\text{sel},\text{next}) =$$

$$\text{concat}((\text{SUBLISTSEQ}(p,q,\text{sel},\text{next}))_i,$$

$$\text{concat}([\text{expr}],(\text{SUBLISTSEQ}(q.\text{next},r,\text{sel},\text{next}))_i))$$

$$\underline{\text{and}} \ (\text{com} \neq \text{sel} \supset$$

$$(\text{SUBLISTSET}(p,r,\text{com},\text{next}) =$$

$$(\text{SUBLISTSET}(p,r,\text{com},\text{next}))_i \ \underline{\text{and}}$$

$$\text{SUBLISTSEQ}(p,r,\text{com},\text{next}) =$$

$$\left.(\text{SUBLISTSEQ}(p,r,\text{com},\text{next}))_i)\right]$$

A more complicated operation is the updating of a link component. In this case, the structure remains a list only if the new value references a list. In addition, the record whose component is being updated cannot initially be ultimately connected to the new value, or else the structure will become circular. The values of the representation functions are changed by adjoining onto the value represented by the initial sublist the value represented by the newly adjoined list:

Lemma I.19

$$[\text{LIST}(p,\text{next}) \ \underline{\text{and}} \ (p \overset{\text{next}}{\Longrightarrow} q \overset{\text{next}}{\Longrightarrow} + \ \underline{\text{null}})]$$

$$\{q.\text{next} := \text{expr}\}$$

$$[(\text{LIST}(p,\text{next}) \ \underline{\text{iff}} \ \text{LIST}(\text{expr},\text{next}) \ \underline{\text{and}} \ \underline{\text{not}}(\text{expr} \overset{\text{next}}{\Longrightarrow} q)) \ \underline{\text{and}}$$

$$(p \overset{\text{next}}{\Longrightarrow} q) \ \underline{\text{and}}$$

$$(q \overset{\text{next}}{\Longrightarrow} + \ r \ \underline{\text{iff}} \ \text{expr} \overset{\text{next}}{\Longrightarrow} r) \ \underline{\text{and}}$$

$$(\text{LIST}(p,\text{next}) \ \underline{\text{and}} \ q \overset{\text{next}}{\Longrightarrow} + \ r) \supset$$

$$\text{SUBLISTSET}(p,r,\text{sel},\text{next}) =$$

$$(\text{SUBLISTSET}(p,(q.\text{next})_i,\text{sel},\text{next}))_i \ \cup$$

$$(\text{SUBLISTSET}(\text{expr},r,\text{sel},\text{next}))_i \ \underline{\text{and}}$$

$$\text{SUBLISTSEQ}(p,r,\text{sel},\text{next}) =$$

$$\text{concat}((\text{SUBLISTSEQ}(p,(q.\text{next})_i,\text{sel},\text{next}))_i,$$

$$(\text{SUBLISTSEQ}(\text{expr},r,\text{sel},\text{next}))_i))]$$

The link component of a record can also be changed by assigning to it a reference to a newly-created record. The conditions for the structure remaining a list are then the same as in the previous lemma. The new values of the representation functions are then also the same, except for the insertion of the value of the new record's component:

Lemma L.20

$$[\text{LIST}(p,\text{next}) \ \underline{\text{and}} \ (p \xRightarrow{\text{next}} q \xRightarrow{\text{next}} + \ \underline{\text{null}})]$$

$$\{q.\text{next} := \text{rclass}(\ldots,\text{sel:val},\ldots,\text{next:expr},\ldots)\}$$

$$[(\text{LIST}(p,\text{next}) \quad \underline{\text{iff}}$$

$$\text{LIST}(\text{expr},\text{next}) \ \underline{\text{and}} \ \underline{\text{not}} \ (\text{expr} \xRightarrow{\text{next}} q)) \ \underline{\text{and}}$$

$$(p \xRightarrow{\text{next}} q) \quad \underline{\text{and}}$$

$$(q \xRightarrow{\text{next}} + \ r \quad \underline{\text{iff}} \quad \text{expr} \xRightarrow{\text{expr}} r) \quad \underline{\text{and}}$$

$$(\text{LIST}(p,\text{next}) \ \underline{\text{and}} \ q \xRightarrow{\text{next}} + \ r \quad \supset$$

$$\text{SUBLISTSET}(p,r,\text{sel},\text{next}) =$$

$$(\text{SUBLISTSET}(p,(q.\text{next})_i,\text{sel},\text{next}))_i \ \cup \ \{\text{val}\}$$

$$\cup \ (\text{SUBLISTSET}(\text{expr},r,\text{sel},\text{next}))_i \quad \underline{\text{and}}$$

$$\text{SUBLISTSEQ}(p,r,\text{sel},\text{next}) =$$

$$\text{concat}((\text{SUBLISTSEQ}(p,(q.\text{next})_i,\text{sel},\text{next}))_i,$$

$$\text{concat}([\text{val}],$$

$$(\text{SUBLISTSEQ}(\text{expr},r,\text{sel},\text{next}))_i))]$$

One specific case of this lemma is important enough to warrant separate statement as a corollary. This is the case where "expr" is in fact "q.next." The effect of the statement is then to simply insert the new record into the list just after the record referenced by q. Several simplifications are then possible, since the structure must remain a list, and the value of the "set" function simply contains one new value.

Corollary L.21

$$[\text{LIST}(p,\text{next}) \ \underline{\text{and}} \ (p \xRightarrow{\text{next}} q \xRightarrow{\text{next}} + \ r)]$$

$$\{q.\text{next} := \text{rclass}(\ldots,\text{sel:val},\ldots,\text{next:q.next},\ldots)\}$$

$$[\text{LIST}(p,\text{next}) \ \underline{\text{and}} \ (p \xRightarrow{\text{next}} q \xRightarrow{\text{next}} + \ r) \ \underline{\text{and}}$$

$$\text{SUBLISTSET}(p,r,\text{sel},\text{next}) =$$

$$(\text{SUBLISTSET}(p,r,\text{sel},\text{next}))_1 \ \cup \ \{\text{val}\} \ \underline{\text{and}}$$

$$\text{SUBLISTSEQ}(p,r,\text{sel},\text{next}) =$$

$$\text{concat}((\text{SUBLISTSEQ}(p,(q.\text{next})_1,\text{sel},\text{next}))_1,$$

$$\text{concat}([\text{val}],(\text{SUBLISTSEQ}((q.\text{next})_1,r,\text{sel},\text{next}))_1))]$$

The final kind of statement to consider is a record creation statement in which a reference to the new record is assigned to a simple variable:

Lemma L.22

$$[\underline{\text{true}}]$$

$$\{p := \text{rclass}(,..,\text{sel:val},\ldots,\text{next:expr},\ldots)\}$$

$$[(\text{LIST}(p,\text{next}) \ \underline{\text{iff}}$$

$$\text{LIST}(\text{expr},\text{next}) \ \underline{\text{and}} \ \underline{\text{not}}(\text{expr} \xRightarrow{\text{next}} + \ p)) \ \underline{\text{and}}$$

$$(p \xRightarrow{\text{next}} + \ r \ \ \underline{\text{iff}} \ \ \text{expr} \xRightarrow{\text{next}} r) \ \underline{\text{and}}$$

$$(\text{LIST}(p,\text{next}) \ \underline{\text{and}} \ \ r \neq p \ \supset$$

$$\text{SUBLISTSET}(p,r,\text{sel},\text{next}) =$$

$$\{\text{val}\} \cup \text{SUBLISTSET}(\text{expr},r,\text{sel},\text{next}))_1 \ \underline{\text{and}}$$

$$\text{SUBLISTSEQ}(p,r,\text{sel},\text{next}) =$$

$$\text{concat}([\text{val}],(\text{SUBLISTSEQ}(\text{expr},r,\text{sel},\text{next}))_1))] \ .$$

Notice that there is no precondition attached to this lemma, or more precisely that the precondition is always "true." This is because it makes no difference what the value of p before the assignment is, since it is completely updated by the assignemnt. Otherwise, this case is quite similar to that of Lemma L.20.

An important part of any proof of correctness is the proof of termination. There are really two separate issues in this: one is that all operations in the program are well-defined; that if "p.sel" appears in a statement, for instance, then "p ≠ null" is guaranteed. This aspect is most easily handled as such statments are encountered in the body of each verification condition. Whenever there is any room for doubt, we will explicitly show that the given reference is not null.

The other aspect involves termination of each loop in the program. It has been shown [Flo67] that a loop is guaranteed to terminate if some integer quantity can be found which is always finite and positive, and which decreases with each iteration through the loop.

The most common form of iteration when operating on a list structure is a loop of the form[1]

"while p ≠ null [and Q] do S"

or "until p ≠ null [or Q] do S",

where Q, if included, is a predicate, and S is the loop

[1] The brackets indicate a clause whose inclusion is optional.

body, either a simple or compound statement. In loops such as these, the length of the list referenced by p is a finite, positive, decreasing quantity, and thus can be used to prove loop termination. To formalize this, the length of a list can be defined:

Definition I.23 LIST(p,next) ➽

list-length(p) = if p = null then 0
else 1 + list-length(p.next).

The following lemma is then obvious:

Lemma I.24 If "LIST(p,next)" is invariant within all iterations of a loop of the form

"while (p ≠ null [and Q]) do S"
or "until (p = null [or Q]) do S",

then the loop terminates provided that the effect of the loop body S on the variable p is to assign to p the value of an expression "expr", such that "$p \overset{next}{\Longrightarrow} +$ expr" was true initially, i.e. at the beginning of that execution of S.

The most common use of this lemma is when "expr" is simply "p.next", since a loop is most often used to iterate through the successive records in a list.

In proving the termination of a program, we will deal exclusively with the proof of loop termination. As discussed above, the well-definedness of component

selection will be handled within the proofs of the v.c.'s, and will thus be considered "given" when termination is proved separately at the end.

Now that we have developed a framework within which to verify programs which operate on lists, we test it out by the straightforward method of putting it into practice for the proofs of actual programs. The three programs proved below vary in complexity, but all are solutions to real-life programming problems. They have been designed to illustrate a broad range of issues associated with proving the correctness of programs, particularly using the approach of this thesis.

All three of these programs use lists whose records are from the simplest possible class of any practical use, namely:

record class intcell(n integer; next ref(intcell))

This should not be taken as an indication that the approach is limited to use with such a simple record class. The reason for employing the class is the same reason for choosing lists as the subject of this chapter: to reduce the amount of extraneous detail to a minimum, and thus allow the important issues to stand out that much more clearly. Incidentally, since records of this class contain only one reference component, "$\Rightarrow$ " can be used in place of "$\overset{\text{next}}{\Longrightarrow}$ ".

D. First example proof—Search and addition

The first example will be an exceedingly simple program, one which searches the records of a list for a given integer, and, if the integer is not found, adds a record containing this integer component to the beginning of the list. The program is listed in Figure 4.3, with the assertions included as comments. The program invariant, which is given in the very first comment, states that variable sp references a list in which no two distinct records have equal "n" components.

The control structure of the program and the placement of the intermediate loop assertion require that four verification conditions be proved in order to verify the correctness of the program. These correspond to the four possible paths between assertions:

(a) The path from the input assertion to the loop assertion, corresponding to lines (1) and (2) of the program text.

(b) The path from the loop assertion back to itself, representing one iteration of the loop, lines (3) and (2) of the text.

(c) The path from the loop assertion to the output assertion, lines (3), (2), (4) and (5).

(d) The path from the input assertion to the output assertion (in the special case that no loop

```
     procedure intlist_insert(j,sp);
     comment program invariant INV = "LIST(sp,next) and
             (∀ q,r)(sp⟹ q⟹+ r⟹+ null ⊃q.n ≠ r.n)",
          that is, no two records in the list have the
               same "n" component;
     record class intcell(n integer; next ref(intcell));
     declare sp,p ref(intcell);
     declare j integer;
     begin
     comment assert INV;
(1)        p := sp;
(2)        while (p ≠ null  and  p.n ≠ j)  do
           comment assert "INV  and  (sp⟹ p⟹+ null)
             and  LISTSET(sp,n,next) =
                  (LISTSET(sp_o,n,next))        and
               j ∉ SUBLISTSET(sp,p.next_o,n,next) ";

(3)             p := p.next;
(4)        if p = null     then
(5)             sp := intcell(j,sp);
     comment  assert "INV  and
            LISTSET(sp,n,next) = (LISTSET(sp_o,n,next))
                                                  U {j}_o";
     end intlist_insert;
```

Figure 4.3

Example Program 1

iterations are performed), lines (1), (2), (4) and (5) of the text.

The body of code associated with verification condition (a) consists of the assignment on line (1) and the test on line (2) with a true outcome. The proof of the v.c. consists of proving the validity of each of the five clauses of the loop assertion[1], following execution of this code, with the initial assumption that the input assertion is valid. Each of the five clauses can be proved in turn:

(i) "LIST(sp,next" is valid since it is part of of the input assertion, and thus by assumption valid initially; and since the body of code contains no assignment to sp or to any link component in the list, so that it could not become invalidated.

(ii) Similarly, "$(\forall q,r)(sp \Rightarrow q \Longrightarrow^{+} r \Longrightarrow^{+} \underline{null} \supset q.n \neq r.n)$" remains valid from the input assertion since no assignment is made to either sp or to any component in the list.

(iii) Also, "$LISTSET(sp,n,next) = (LISTSET(sp_0,n,next))_0$" is valid, since it is valid initially (trivially) and since no assignment is made which could invalidate it.

---

[1]Since the program invariant contains two clauses, the total number in the loop assertion is five.

(iv) The assignment on line (1) sets p to the value of sp, so it follows directly that "sp$\Rightarrow$ p" is valid, since "$\Rightarrow$ " is reflexive. And, since "sp$\Rightarrow$ null" by definition of LIST, "p$\Rightarrow$ null" also follows. Finally, the test on line (2) confirms that p $\neq$ null, so that "sp$\Rightarrow$ p$\Rightarrow\!\!\!\!\neq$null" is validated.

(v) SUBLISTSET(sp,p.next,n,next) = $\{$sp.n$\}$ since sp = p, by Corollary I.13. By the test on line (2), sp.n $\neq$ j, so it follows that "j $\notin$ SUBLISTSET(sp,p.next,n,next)."

Verification condition (b) concerns the loop body, i.e. the assignment on line (3) followed by the test on line (2) with an outcome of true. Its proof is also in five parts, corresponding to the five clauses in the loop assertion:

(i) "LIST(sp,next)" and

(ii) "$(\forall q,r)$(sp$\Rightarrow$ q$\Rightarrow+$ r$\Rightarrow+$ null $\supset$ q.n $\neq$ r.n)" and

(iii) "LISTSET(sp,n,next) = (LISTSET(sp_0,n,next))$_0$" are all valid because they are valid initially, and since no assignment is made in the body to sp or to any component in the list which might invalidate them.

(iv) Let the value of p initial to the v.c. be called

"p_1". Then the assignment on line (3) updates the value of p to p_1.next. From the initial validity of the loop assertion, "$p_1 \Longrightarrow +$ null" so this selection is well-defined, and further "$p \Longrightarrow$ null" is valid from the definition of "$\Longrightarrow$". Also from the initial assertion, $sp \Longrightarrow p_1$ so $sp \Longrightarrow p_1.next = p$. Finally, the test on line (2) confirms that $p \neq$ null, so this validates the clause "$sp \Longrightarrow p \Longrightarrow +$ null."

(v) From the definition of SUBLISTSET, it is clear that SUBLISTSET(sp,p.next,n,next) = SUBLISTSET(sp,p,n,next) $\cup$ $\{p.n\}$. The initial assertion ensures that $j \notin$ SUBLISTSET(sp,p,n,next) and the test on line (2) confirms that $j \neq p.n$, so "$j \notin$ SUBLISTSET(sp,p.next,n,next)" is validated.

The body of code for verification condition (c) contains the assignment on line (3), the test on line (2) with false outcome, and lines (4) and (5). The false outcome of the test on line (2) can be for one of two reasons, depending upon which of the two clauses in the test is false. The proof of this v.c. is by case analysis of these two possibilities:

(i) If the test outcome is because of the first clause, then p = null after execution of

line (3). This means that p_i.next = null, where "p_i" is the initial value of p to this v.c. Then the test on line (4) is true, and the assignment on line (5) is performed. "LIST(sp,next)" is finally valid, by Lemma I.22. By the same lemma, LISTSET(sp,n,next) = $\{j\}$ U (LISTSET(sp_o,n,next))$_o$, since the initial validity of the loop assertion ensures that this value has not changed. Furthermore, since p_i.next = null, it follows that SUBLISTSET(sp,p_i.next,n,next) = LISTSET(sp,n,next), so by the initial assertion again, $j \notin$ LISTSET(sp,n,next)$_i$, which can be rephrased, $(\forall q)(sp_i \Longrightarrow q \supset q.n \neq j)$. This, together with the fact that the only change to any "n" component in the list is the addition of the new record with "n" component equal to j, means that the initial validity of "$(\forall q,r)(sp \Longrightarrow q \Longrightarrow^+ r \Longrightarrow^+ \text{null} \supset q.n \neq r.n)$" is sufficient to verify its final validity.

(ii) If the test outcome is due to the second clause, then p $\neq$ null and p.n = j (the selection is well-defined since p $\neq$ null). Then the test on line (4) is false, and the assignment on line (5) is skipped. Since "LIST(sp,next)" is valid initially, it must be valid finally, as

no action is performed which could invalidate it. Since $sp \Longrightarrow p_1 \Longrightarrow^+ \underline{null}$, "$sp \Longrightarrow p$" follows in the same way that it did for verification condition (b), and so by Corollary L.15, $j \in$ LISTSET(sp,n,next) which by the initial assertion is equal to $(\text{LISTSET}(sp_o,n,next))_o$. Thus, "$\text{LISTSET}(sp,n,next) = ((\text{LISTSET}(sp_o,n,next))_o \cup \{j\}$". Finally, since no record components in the list are changed, "$(\forall q,r)(sp \Longrightarrow q \Longrightarrow^+ r \Longrightarrow^+ \underline{null} \supset q.n \neq r.n$" remains valid.

Verification condition (d) is associated with the special case that no loop iterations are performed, i.e. that the test on line (2) is *false* initially. The body of code thus consists of line (1), the *false* test on line (2), and lines (4) and (5). From the action of line (1), p = sp. The proof is again by case analysis on whether the *false* outcome is due to the first or second clause of the test:

(i) If due to the first clause, then $sp_o = \underline{null}$. Then $(\text{LISTSET}(sp_o,n,next))_o = \emptyset$. Also, then the test on line (4) is *true*, and the assignment on line (5) is made. This results in sp.n = j and sp.next = *null*. "LIST(sp,next)" and "$\text{LISTSET}(sp,n,next) = (\text{LISTSET}(sp_o,n,next))_o \cup \{j\}$" then follow directly from the definitions.

Because there is only the one record in the list, "$(\forall q,r)(sp \Longrightarrow q \Longrightarrow^{+} r \Longrightarrow^{+} \underline{null} \supset q.n \neq r.n)$" is true trivially.

(ii) If due to the second clause, then sp.n = j. Then the test on line (4) is <u>false</u> and the action on line (5) is not performed. No change is made to the list, so both clauses of the invariant remain valid, and $LISTSET(sp,n,next) = (LISTSET(sp_0,n,next))_0$. From Corollary L.15, $j \in LISTSET(sp,n,next)$, so "$LISTSET(sp,n,next) = (LISTSET(sp_0,n,next))_0 \cup \{j\}$" is valid.

The final aspect of the proof involves proof of loop termination. The loop body consists of the single statement "p := p.next" on line (3), and since "LIST(sp,next)" and "$sp \Longrightarrow p$" are invariant within the loop, "LIST(p,next)" is always true in the loop, so the conditions for Lemma L.24 are met. By direct application of the lemma then, the loop is guaranteed to terminate. Thus, the program terminates.

E. Issues raised by the example proof

While the program is quite simple and the proof relatively straightforward, there are several instructive points raised by the proof above. First, despite the

simplicity of the program, as well as the use of a number of lemmas and corollaries developed earlier, the size of the proof is disconcertingly large. Unfortunately, proofs of programs tend to be quite a bit longer than one might hope. In fact, the length and complexity of the proofs, even of the simplest programs, has been the major obstacle in the way of building automatic program verification systems, a goal towards which much effort has been expended with rather disappointing results. In those systems which have been built, the theorem prover has turned out to be the weak link (see, for example [Kin69], [Man71], etc.).

Because of this size problem, it is helpful to the person proving programs to do everything possible to shorten and simplify the proofs. In fact, we could have significantly reduced the size of the above proof by a very simple technique. Consider Figure 4.4, which contains a flowchart of the example program. Notice that the loop assertion has been moved to a different control point, viz. before the loop termination test on line (2) rather than after it.[1] This change, though small on the

---

[1] This change of location would necessitate two changes to the loop assertion itself. The clause "(sp⟹ p⟹+ null)" must be changed to "(sp⟹ p⟹ null)", and the last clause must be changed to "j ∉ SUBLISTSET(sp,p,n,next)". These two changes reflect the fact that the test on line (2) can no longer be assumed to have a true outcome.

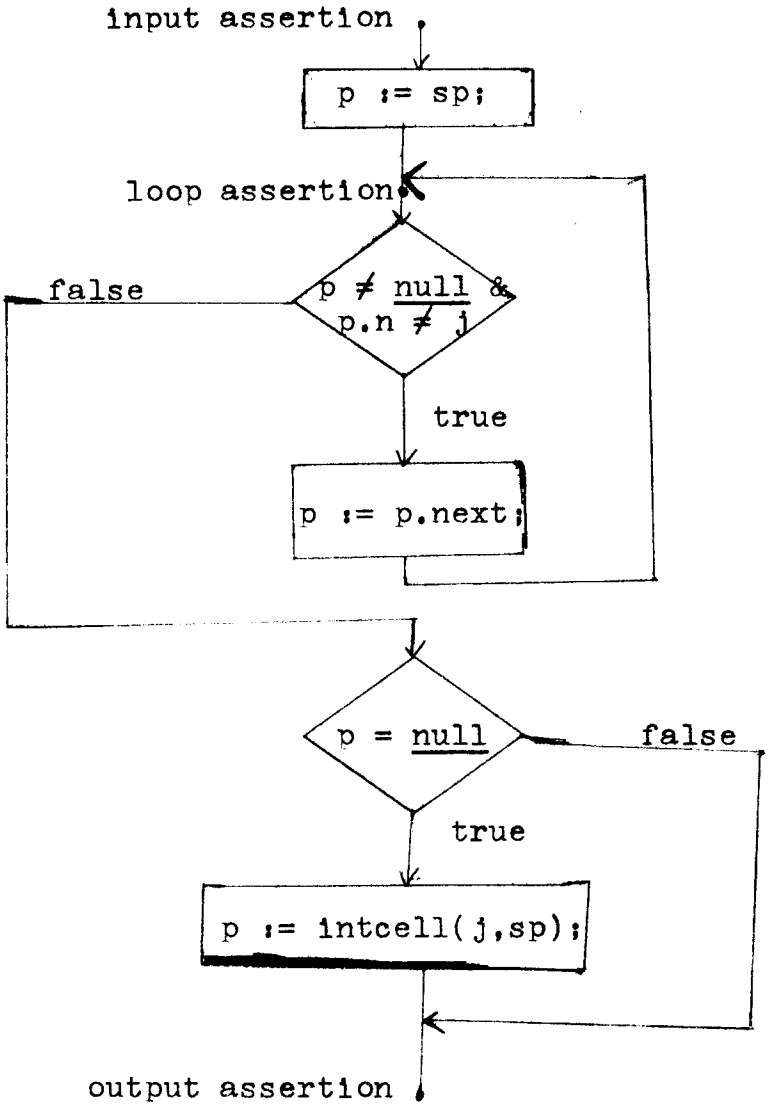


Figure 4.4

Flowchart of example program 1

surface, has rather important repercussions. First, the number of v.c.'s is diminished from four to three. The fourth v.c.(d) in the original proof is no longer needed because the path from the input assertion to the output assertion now includes the loop assertion; v.c.(d) degenerates into the composition of v.c.'s (a) and (c). In addition, v.c.(c) is now simpler because the loop body is no longer part of its body. The improvement here is slight because the loop body consists of only one statement, line (3). However, in the general case, where the loop body is of substantial size, the change can be significant.

Why did we not place the loop assertion at this control point originally? There are two reasons: one is that the change was not obvious when we were dealing with the program text. It took the pictorial representation of the flowchart to make clear the benefit this change would bring. The other reason is that it is not immediately obvious where in the source text this particular control point represents. If we put the assertion between lines (1) and (2), it is not apparent that we intend this point to be part of the loop.[1] For both of these reasons, programs will henceforth be translated to flowchart form

[1] The reason for this difficulty, of course, is purely due to the syntax of the source language. Deutsch [Deu73] solved this problem by using a loop statement to mark the beginning of the loop, before the while statement, in his language.

before constructing and placing intermediate assertions.

The next important point is that the loop assertion consists of the minimum information necessary for the proof, given the input and output assertions. Each clause of the loop assertion is used in v.c.(c), so that the validity of the output assertion could not be verified without all the information the loop assertion contains. (It might also be necessary, though it is not so here, for certain information to be included in the loop assertion so as to prove the continuing validity of other parts of the loop assertion itself, i.e. to prove v.c. (b).) Because of the lengthiness of the proofs, it is important that the intermediate assertions of a program contain no more information than is necessary for the proof. The introduction of a new clause in an intermediate assertion increases the size of the proof by adding that clause to what must be proved in at least two v.c.'s.

Finally, it was necessary in the proof to constantly use statements like "Since 'LIST(sp,next)' is valid initially, and nothing in the body of source code changes either sp or any link in the list, it must still be valid." The problem with such statements is not any imprecision about them, but rather their wordiness. It makes more sense to have an additional lemma which states the same result, and which can be referred to whenever it is needed.

That is the reason for the following:

Lemma L.25 (i) If "LIST(p,next)" is valid before a body of code is executed, and if no assignment is made in that body of code to either p, or to any expression equivalent[1] to q.next, for some q such that "$p \xRightarrow{next} q$" is also valid initially, then "LIST(p,next) is true after execution of the code.

(ii) If "$p \xRightarrow{next} r$" is valid before a body of code is executed, and if no assignment is made in that body of code to either p, or to any expression equivalent to q.next, for some q such that $p \xRightarrow{next} q \xRightarrow{next} + r$ initially, then "$p \xRightarrow{next} r$" is true after execution of the body of code.

(iii) If "LIST(p,next)" and "$p \xRightarrow{next} r$" are valid before a body of code is executed, and if no assignment is made in that body of code to either p, or to any expression equivalent to q.sel or q.next, for some q such that $p \xRightarrow{next} q \xRightarrow{next} + r$ initially, then the values of SUBLISTSET(p,r,sel,next) and SUBLISTSEQ(p,r,sel,next) are left unchanged by the execution of the body of code.

---

[1] "Equivalent" here means "equal in L-value" (i.e. address) as opposed to "equal in R-value"—the "=" relation. That is, the "next" component of some record is updated.

F. Second example proof—List reversal

The second example is slightly more complicated than the first, due to a number of factors. For one thing, the program involves changing link values in the list, something which the first program did not do. For this reason, it provides a more stringent test of the formalism. It also performs a more complicated operation—reversing a list. This means that the abstract operation which the computation corresponds to is one on sequences, which are more complicated than sets. Finally, it involves manipulations on two different list structures simultaneously in the course of its operation.

Since the program corresponds to the abstract operation of sequence reversal, a suitable formal definition of this operation is required. The definition can be constructed using the primitive operations on sequences—first, final, last, initial, and concat—as axiomatized by Hoare [Hoa72a] and repeated in Appendix A:

reverse(s) = if s = [] then []
else concat([last(s)],reverse(initial(s))).

The abstract operation "reverse" can now be used in assertions and manipulated in accordance with the above definition. This technique of defining new abstract operations in terms of the primitives available, and

then using the new operations to express results of computational manipulations, is an important and powerful technique in our repertoire.

The program appears in Figure 4.5, with its antecedent and consequent stated. The corresponding flowchart, with the intermediate assertion attached, is in Figure 4.6. Figure 4.7 illustrates the intermediate state of the computation at the end of a loop iteration at line (8). The list of records referenced by lp has been reversed, while those referenced by mp (and also rp) are unchanged. A single loop iteration takes one additional record from the head of the "unchanged" list and places it at the head of the "reversed" list. The loop terminates when the unchanged list has been exhausted.

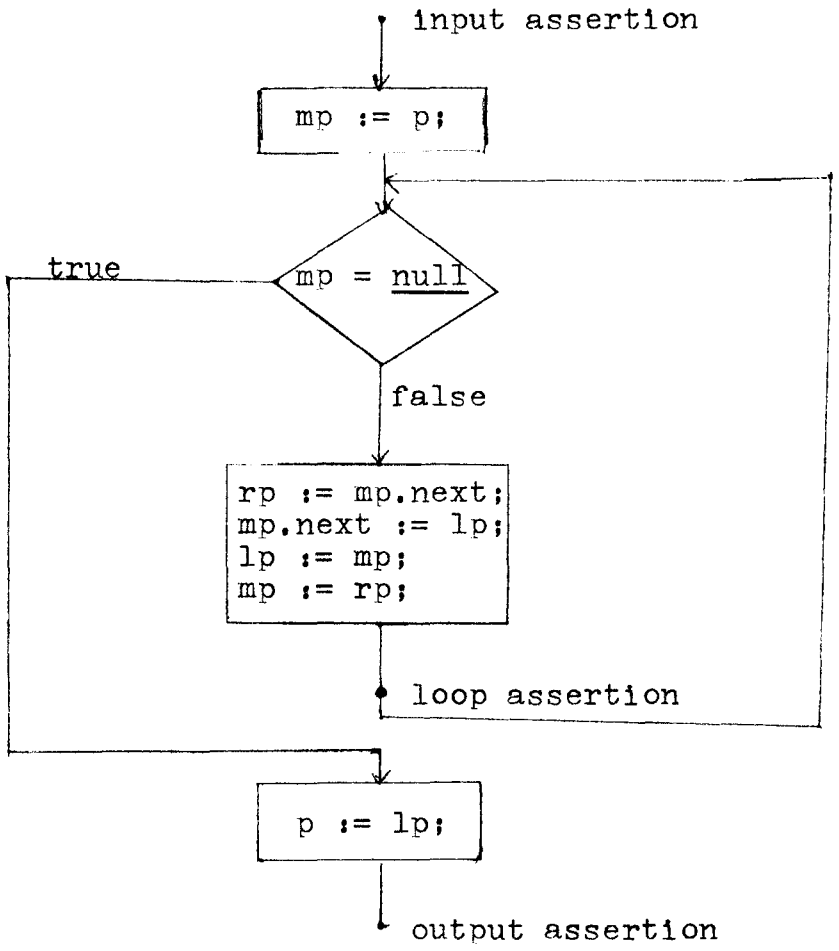
No matter how general the rule, there will always be exceptions: having just demonstrated that the number of verification conditions for a single-loop program can be reduced from four to three by judicious placement of the loop assertion, I now locate the loop assertion so as to require four v.c.'s. Why?

The reason is that the loop assertion itself would have to be made more complex in order to locate it at its "optimal" (in terms of number of v.c.'s) location, just before the loop termination test on line (2). The difference between the "optimal" and actual placements

```
     procedure intlist_reverse(p);
     comment program invariant is "LIST(p,next)";
     record class intcell(n integer; next ref(intcell));
     declare p,lp,mp,rp ref(intcell));
     begin
     comment  assert "LIST(p,next)";
(1)       mp := p;
(2)       until mp = null   do
(3)            begin
(4)            rp := mp.next;
(5)            mp.next := lp;
(6)            lp := mp;
(7)            mp := rp;
(8)            end;
(9)       p := lp;
     comment  assert "LIST(p,next)   and
       LISTSEQ(p,n,next) = reverse((LISTSEQ(p_o,n,next))_o)";
     end intlist_reverse;
```

Figure 4.5

Example program 2



loop assertion: "LIST(p,next) and LIST(mp,next) and
mp = $(lp.next)_o$ and
LISTSEQ(mp,n,next) = $(LISTSEQ(mp,n,next))_o$ and
LISTSEQ(lp,n,next) =
reverse($(SUBLISTSEQ(p,lp.next_o,n,next))_o$)"

Figure 4.6

Flowchart of example program 2

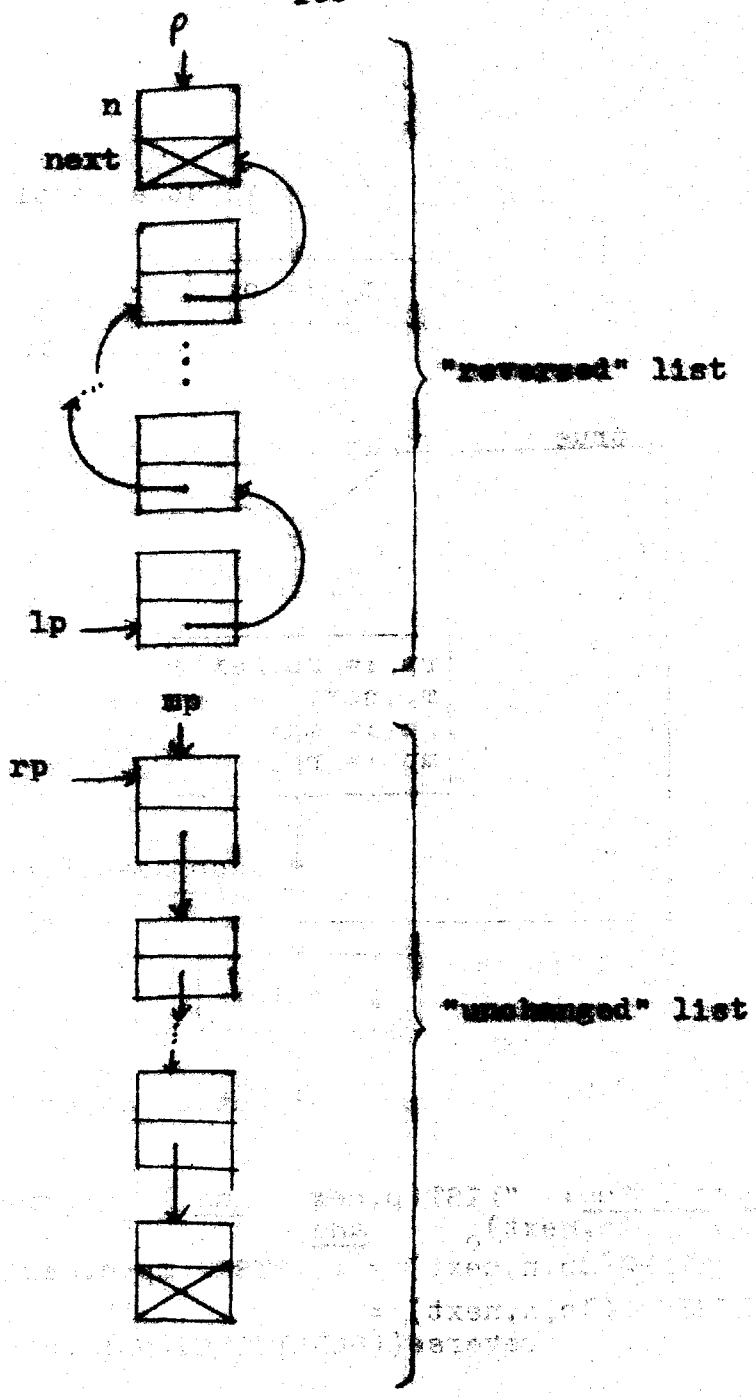


Figure 4.7

Intermediate computation state of example program 2.
(Original order of records is top to bottom.)

is simply that the loop assertion reflects at least one execution of the loop body. Otherwise, the two locations are identical. In order for the loop assertion to fit the optimal location, it would also have to apply to the case of no previous iterations; this change would require a long, two-case expression which would complicate the proof of the v.c. whose body is one loop iteration. For this reason, the goal of overall proof simplicity is best met by the location for the assertion given in the flow-chart. It should be noted that locating the assertion at the end of the loop body rather than the beginning does simplify the v.c. whose path is between the loop assertion and the output assertion, just as the optimal location would have done. Also, the additional v.c. required by this placement is quite simple, since it covers the trivial case of a zero-element (i.e. null) list.

We must thus prove four verification conditions, with paths analagous to those for the first example program.

(a) The first v.c. has the input assertion as its initial assertion, lines (1) through (7) as its body, and the loop assertion as its final assertion. The loop body contains five clauses to be proved:

(i) The effect of the body on variable lp is to assign to it the value of mp on line (6).

From line (1), mp is given the value of p, i.e. a reference to the first record in the list referenced by p (note "LIST(p,next)" in the input assertion, and the false outcome of the test on line (2) ensures that p ≠ null). Line (5) assigns to mp.next the value of lp, which by default is null. So after line (7), the first time through the loop, lp.n = p.n and lp.next = null. From this,"LIST(lp,next)" follows directly from its definition.

(ii) Similarly, "LISTSEQ(lp,n,next) = reverse(SUBLISTSEQ(p,lp.next$_o$,n,next)$_o$)" is immediately true from the definitions of "reverse" and SUBLISTSEQ, and the fact that lp = p.

(iii) Line (7) assigns to mp the value of rp, which by lines (1) and (4) is (p.next)$_o$. This is well-defined, since "LIST(p,next)" is in the input assertion, and p ≠ null by the test on line (2). Since no action is performed on any records in the list beyond thefirst one, i.e. by Lemma L.25, "LIST(mp,next)" remains valid. (It is initially valid, of course, since "LIST(p,next)" is in the input assertion and by Corollary L.4.)

(iv) Similarly, "LISTSEQ(mp,n,next) =

$(LISTSEQ(mp,n,next))_o$" is valid by Lemma I.25.

(v) Since the final value of lp is p_o, as explained in (i), and that of mp is $(p.next)_o$, as exaplained in (iii), "mp = $(lp.next)_o$" is valid.

(b) The second v.c. has the loop assertion as both initial and final assertion, and lines (2) through (7) as body of code. Again, there are five clauses to prove:

(i) From lines (5) and (6), the value of lp.next is "lp_i", the value of lp initial to the v.c. Since "LIST(lp,next)" is valid initially, "LIST(lp,next)" follows directly from Definition I.1 and Theorem I.3.

(ii) By Corollary I.17,
lp.n = last($(SUBLISTSEQ(p,lp.next_o,n,next))_o$).
(Note that from the initial validity of "LISTSEQ(mp,n,next) = $(LISTSEQ(mp,n,next))_o$", this value has not been changed.) By the same corollary, $(SUBLISTSEQ(p,lp,n,next))_o$ = initial($(SUBLISTSEQ(p,lp.next_o,n,next))_o$).
By Definition L.7, it is clear that
LISTSEQ(lp,n,next) =
concat([lp.n] , LISTSEQ(lp.next,n,next)).
Since lp.next = lp_i, the initial validity of the loop assertion says that

LISTSEQ(lp.next,n,next) = LISTSEQ(lp_1,n,next)

= reverse((SUBLISTSEQ(p,lp_1.$next_o$,n,next))$_o$)

= reverse((SUBLISTSEQ(p,lp,n,next))$_o$).

Thus, the statement above,

"LISTSEQ(lp,n,next) =

concat([lp.n] , LISTSEQ(lp.next,n,next))"

becomes

"LISTSEQ(lp,n,next) = concat([last(S)],

reverse(initial(S)))"

where S = (SUBLISTSEQ(p,lp.$next_o$,n,next))$_o$.

But this is just the definition of "reverse,"

so that "LISTSEQ(lp,n,next) =

reverse((SUBLISTSEQ(p,lp.$next_o$,n,next))$_o$)"

is verified.

(iii) From lines (4) and (7), the new value of mp is mp_1.next, where "mp_1" is its value initial to this v.c. Since "LIST(mp,next)" is true initially, it must be true finally by Corollary L.4 and Lemma L.25.

(iv) Also by Lemma L.25,

"LISTSEQ(mp,n,next) = (LISTSEQ(mp,n,next))$_o$"

must remain valid.

(v) By line (6), lp = mp_1 and from (iii) above, mp = mp_1.next, so that "lp = (mp.next)$_1$" is clearly valid. Since "LISTSEQ(mp,n,next) =

$(\text{LISTSEQ}(mp,n,next))_0$" is valid initially, $(mp.next)_1$ must equal $(mp.next)_0$, so that "$lp = (mp.next)_0$" is verified.

(c) The third v.c. takes the loop assertion as its initial assertion, the <u>true</u> test on line (2) followed by line (9) as its body, and the output assertion as its final assertion. There are two clauses to be proved:

(i) "LIST(p,next)" follows directly from the assignment "p := lp" on line (9) and the inclusion of "LIST(lp,next)" in the loop assertion.

(ii) From the <u>true</u> outcome on line (2), mp = <u>null</u>. By the loop assertion, this means that $(lp.next)_0$ = <u>null</u>, so again by the loop assertion,

$$\text{LISTSEQ}(lp,n,next) = \text{reverse}((\text{SUBLISTSEQ}(p_0,\underline{\text{null}},n,next))_0) = \text{reverse}((\text{LISTSEQ}(p_0,n,next))_0)$$

by Corollary L.11. Since p = lp from line (9), this verifies that "$\text{LISTSEQ}(p,n,next) = \text{reverse}((\text{LISTSEQ}(p_0,n,next))_0)$".

(d) The fourth v.c. concerns the special cae of no loop iterations. Its initial and final assertions are the program's input and output assertions, respectively,

and its body of code consists of lines (1), (2) (with a true outcome), and (9). The true outcome means that mp = null, which from line (1) means that p_0 = null. Then $(\text{LISTSEQ}(p_0,n,\text{next}))_0 = [\,]$, the empty sequence, by definition of LISTSEQ. The assignment on line (9) assigns to p the value of lp, which is null by default. So "LIST(p,next)" follows trivially, as does "LISTSEQ(p,n,next) = reverse($(\text{LISTSEQ}(p_0,n,\text{next}))_0$)", since $[\,]$ = reverse($[\,]$) by the definition of "reverse".

By clause (iii) in the proof of v.c.(b), each iteration of the loop assigns to mp the old value of mp.next. Since "LIST(mp,next)" is valid throughout the loop, Lemma I.24 can be used to establish the termination of the loop, and thus of the program.

Again, notice that the loop assertion is minimal. Each clause in the assertion is required either for the proof of the output assertion in v.c.(c), or else for the proof of the continuing validity of other clauses in the loop assertion in v.c.(b).

Before leaving this example, I would like to extend a word of acknowledgement to Doug McIlroy. In a talk he gave at Project MAC on "What Makes Programs Intelligible", he used this basic program to illustrate the benefit of allowing parallel assignment in programming languages. After hearing his talk, I decided that this would be an interesting program to prove.

G. Third example proof—Insertion sort

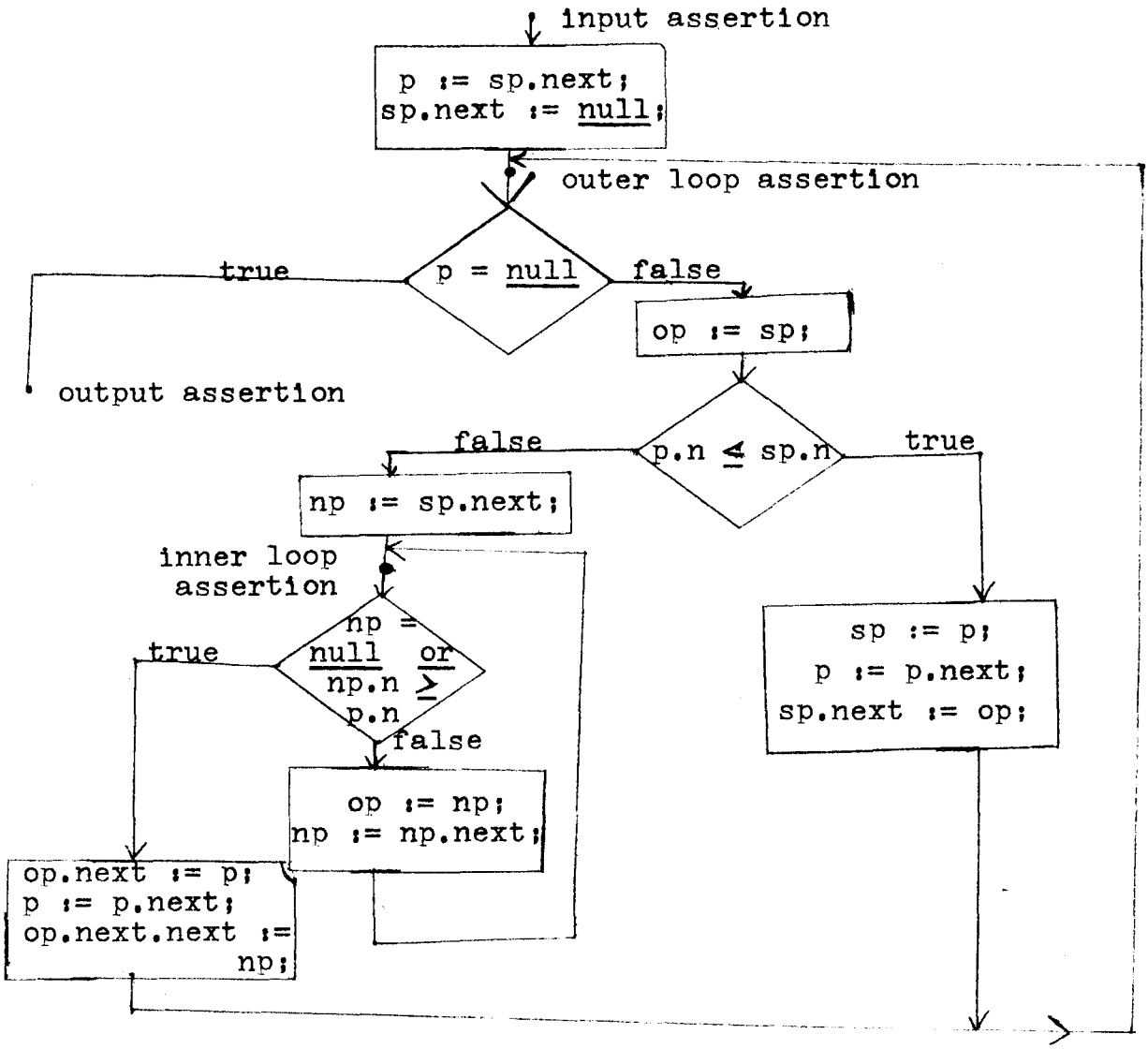
Both of the first two example programs had the standard single-loop control structure. That is, they were each composed of an initialization section, a main section which was a loop driven by a variable iterating over consecutive records in a list, and a final (possibly empty) "clean-up" section. This is a natural control structure to employ for many operations on lists, and in fact is, the one most commonly used for such applications. Some operations require more complicated control structures, however, and proofs of programs with these more complex structures illustrate some ideas which the simpler flow of control does not bring out. The final example is chosen with this in mind.

The program is listed in Figure 4.8. Its purpose is to sort the elements of a list into increasing order of a particular component. The sorting method used is an insertion sort. That is, the program handles two lists, the unsorted input list and the sorted output list. Records are detached from the input list one by one, and inserted into the output list in the correct position to maintain the sorted order. Two loops are required in the program: the outer loop iterates over the records in the input list, and the inner loop iterates over the records thus far in the output list until the proper place for the

```
      procedure intlist_sort(sp);
      comment  program invariant is "LIST(sp,next)"
      record class intcell(n integer; next ref (intcell));
      declare sp,p,op,np  ref(intcell);
      begin
      comment  assert "LIST(sp,next)";
(1)         p := sp.next;
(2)         sp.next := null;
(3)         until p = null    do
(4)              begin
(5)              op := sp;
(6)              if  p.n ≤ sp.n    then
(7)                   begin
(8)                   sp := p;
(9)                   p := p.next;
(10)                  sp.next := op;
(11)                  end;
(12)             else begin
(13)                  np := sp.next;
(14)                  until (np = null or np.n ≥ p.n) do
(15)                        begin
(16)                        op := np;
(17)                        np := np.next;
(18)                        end;
(19)                  op.next := p;
(20)                  p := p.next;
(21)                  op.next.next := np;
(22)                  end;
(23)             end;
      comment assert "LIST(sp,next)  and
           (sp⇒q⟹+ r⟹+ null ⊃ q.n ≤ r.n)  and
           LISTSET(sp,n,next) = (LISTSET(sp_o,n,next))_o";
      end intlist_sort;
```

Figure 4.8

Example program 3



outer loop assertion: "LIST(sp,next) and LIST(p,next) and
(sp⇒q⇒+ r⇒+ null ⊃ q.n ≤ r.n) and
LISTSET(sp,n,next) U LISTSET(p,n,next) =
$(\text{LISTSET}(sp_0,n,next))_0$"

inner loop assertion: "(outer loop assertion) and
(p ≠ null) and (sp⇒ op→ np) and (p.n > op.n)"

Figure 4.9

Flowchart of example program 3

insertion is found. The flowchart in Figure 4.9 illustrates the control structure of the program, and contains both of the intermediate assertions required.

Notice that there are two possible paths through the body of the outer loop, depending upon the outcome of the test on line (6). With a true outcome, lines (7) through (11) are executed. If the outcome is false, the alternative action is lines (12) through (22), including an unspecified number of iterations through the loop of lines (14) through (18). In the first example program, there was a conditional statement, but we did not employ a separate verification condition for each outcome due to its simplicity. Instead, the proof was handled in a single v.c. which was proved by cases. This situation is more complex, though, so separate v.c.'s are used. There are thus six verification conditions, as indicated by the following table:

v.c.	initial assertion	body	final assertion
(a)	input	(1)-(2)	outer loop
(b)	outer loop	(3)-(11)	outer loop
(c)	outer loop	(3)-(6), (12)-(13)	inner loop
(d)	inner loop	(14)-(18)	inner loop
(e)	inner loop	(14), (19)-(22)	outer loop
(f)	outer loop	(3)	output

This illustrates the tremendous benefit obtainable by judicious location of intermediate assertions. Placement of both loop assertions after their corresponding tests, rather than before, would cause an increase in the number of v.c.'s from six to ten, as well as more complicated bodies for some v.c.'s. The proof then involves proving the six v.c.'s:

(a) The body of this v.c. consists simply of the two statements, lines (1) and (2). From their action, $p = (sp_0.next)_0$ and sp.next = null. There are four clauses in the outer loop assertion:

(i) "LIST(sp,next)" follows directly from the definition of LIST.

(ii) "LIST(p,next)" follows directly from the fact that $p = (sp_0.next)_0$ and the initial validity of "LIST(sp,next)" using Corollary L.4 and Lemma L.25.

(iii) "$(sp \Longrightarrow q \Longrightarrow^{+} r \Longrightarrow^{+} null \supset q.n \leq r.n)$" is valid trivially. Since sp.next = null, there exist no q and r which satisfy the hypothesis of the implication.

(iv) $LISTSET(sp,n,next) \cup LISTSET(p,n,next) = \{sp_0.n\} \cup LISTSET(sp_0.next_0,n,next) = (LISTSET(sp_0,n,next))_0$

directly from the definition of LISTSET.

(b) This v.c. covers the case that the test on line (6) has a true outcome, so that lines (7) through (11) are executed and control loops back to the point where the outer loop assertion is located. The executed statements of the body are the test on line (3), the assignment on line (5), the test on line (6), and the assignment statements on lines (8), (9), and (10). There are four clauses in the outer loop assertion to be proved:

(i) The final value of sp is p_i, from line (8). Line (10) assigns to sp.next the value of op, which is sp_i by line (5). Since "LIST(sp,next)" is valid initially, sp.next references a list, so by Definition L.1 and Theorem I.3, "LIST(sp,next)" is valid finally.

(ii) The sole statement affecting p is line (9), assigning to p the value of p_i.next. By the test on line (3), $p_i \neq$ null, so the result is well-defined, and by Corollary L.4, "LIST(p,next)" is valid.

(iii) If "sp$\Rightarrow$ q$\Longrightarrow$+ r$\Longrightarrow$+ null", then one of two things must be true of the record referenced by q: either it was in the initial list, or it was the record added to the list by line (10), i.e. it was initially referenced by p_i. If

the former, then for any reference r such that $q \Longrightarrow^{+} r$, this was also true initially, so that $q.n \leq r.n$ by the initial validity of this clause. If the latter, then by the test on line (6), $q.n \leq sp_i.n$, and by the transitivity of "$\leq$", this means that for any r such that $q \Longrightarrow^{+} r$, $q.n \leq r.n$. In either case, then, the validity of "$(sp \Longrightarrow q \Longrightarrow^{+} r \Longrightarrow^{+} \underline{null} \supset q.n \leq r.n)$" is maintained.

(iv) Since $sp.next = sp_i$, it follows from the definition of LISTSET that the final value of LISTSET(sp,n,next) is simply its initial value augmented by the new value of sp.n, which is $p_i.n$. Since $p = p_i.next$, on the other hand, the final value of LISTSET(p,n,next) is simply its initial value with the (possible) removal of $p_i.n$ (the value is not removed if it is duplicated in the list). In any case, the value of LISTSET(sp,n,next) U LISTSET(p,n,next) is unchanged, so "LISTSET(sp,n,next) U LISTSET(p,n,next) = $(\text{LISTSET}(sp_0,n,next))_0$" remains valid.

(c) The only assignment statements in the body of this v.c. are those on lines (5) and (13). The first of

these is sandwiched between two tests with <u>false</u> outcomes, on lines (3) and (6).

(i) "LIST(sp,next)" and

(ii) "LIST(p,next)" and

(iii) "(sp$\Rightarrow$ q$\Longrightarrow$+ r$\Longrightarrow$+ <u>null</u> $\supset$ q.n $\leq$ r.n)" and

(iv) "LISTSET(sp,n,next) U LISTSET(p,n,next) = (LISTSET(sp_o,n,next))$_o$"

are all valid finally because they are valid initially, and neither assignment in the body affects either list, so by Lemma L.25.

(v) "p $\neq$ <u>null</u>" is valid by the test on line (3).

(vi) From line (5), op = sp and from line (13), np = sp.next, so "sp$\Rightarrow$ op$\rightarrow$ np" follows directly from the definitions.

(vii) Since op = sp, the test on line (6) ensures the validity of "p.n > op.n".

(d) The body of this v.c. is simply the test on line (14) and the two assignment statements on lines (16) and (17).

(i) "LIST(sp,next)" and

(ii) "LIST(p,next)" and

(iii) "(sp$\Rightarrow$ q$\Longrightarrow$+ r$\Longrightarrow$+ <u>null</u> $\supset$ q.n $\leq$ r.n)" and

(iv) "LISTSET(sp,n,next) U LISTSET(p,n,next) = (LISTSET(sp_o,n,next))$_o$" and

(v) "p ≠ null"

are again all valid because of their initial validity, and by Lemma I.25.

(vi) From line (16), op = np_i, while from line (17), np = np_i.next. (By the test on line (14), np_i ≠ null, so this is well-defined.) So op→ np, and since sp⇒ np_i from the initial validity of this clause, the final validity of "sp⇒ op→ np" is verified.

(vii) Since op = np_i, the test on line (14) ensures the validity of "p.n > op.n".

(e) The body of this v.c. consists of a true outcome on line (14), and the assignments on lines (19), (20), and (21). There are four clauses to be proved:

(i) Since "LIST(sp,next)" and "sp⇒ op→ np" are valid initially, np references a list, by Corollary I.4. Since by line (21), op.next.next = np, this means that op.next.next references a list, so op.next references a list, and so op references a list (all by Definition L.1). Finally, since sp⇒ op, this means that sp references a list, so "LIST(sp,next)" is valid.

(ii) Since "LIST(p,next)" and "p ≠ null" are valid initially, the effect of line (20) leaves "LIST(p,next)" valid.

(iii) Initially, op.next = np, so since sp⟹ op, initially LISTSET(sp,n,next) = SUBLISTSET(sp,np,n,next) U LISTSET(np,n,next) by Corollaries L.12 and L.10. The effect of line (19) is to assign to op.next the value of p_i, but line (21) assigns to p_i.next the value of np, so

SUBLISTSET(sp,np,n,next) = $\{p_i.n\}$ U (SUBLISTSET(sp,np,n,next))$_i$,

i.e. its initial value augmented by the new element p_i.n. The value of LISTSET(np,n,next) is unchanged, by Lemma L.25. So, the value of LISTSET(sp,n,next), which is the union of these two, is its old value augmented by the new element p_i.n. On the other hand, p is assigned the value of p_i.next by line (20), so the value of LISTSET(p,n,next) is its initial value, with the (possible) removal of p_i.n. (As in v.c.(b), the value is not removed if it is duplicated in the list.) The value of LISTSET(sp,n,next) U LISTSET(p,n,next) is thus unchanged, and so the validity of

"LISTSET(sp,n,next) U LISTSET(p,n,next) =

(LISTSET(sp_o,n,next))$_o$"

is preserved.

(iv) If "(sp$\Longrightarrow$ q$\Longrightarrow$+ r$\Longrightarrow$+ null)" is true, then either one of q and r is equal to p_i, or neither is. If neither, then q.n $\leq$ r.n by the initial validity of the inner loop assertion. If q is equal to p_i, then q.n $\leq$ np.n by the test on line (14) (note that np $\neq$ null if such an r exists), so since q$\rightarrow$ np (see (iii) above), np$\Longrightarrow$ r. By the initial assertion, np.n $\leq$ r.n, so by transitivity, q.n $\leq$ r.n. If it is r that equals p_i, then by completely analagous reasoning (using the initial validity of "p.n > op.n" instead of the test on line (14)), q.n $\leq$ r.n can be proved. Thus, "(sp$\Longrightarrow$ q$\Longrightarrow$+ r$\Longrightarrow$+ null $\supset$ q.n $\leq$ r.n)" is verified.

(f) The "body" of this v.c. is simply the true outcome of the test on line (3). There are three clauses in the output assertion to prove:

(i) "LIST(sp,next)" and

(ii) "(sp$\Longrightarrow$ q$\Longrightarrow$+ r$\Longrightarrow$+ null $\supset$ q.n $\leq$ r.n)"

are valid because they are in the initial assertion of the v.c.

(iii) Since p = null by the test on line (3), LISTSET(p,n,next) = $\emptyset$, by the definition of LISTSET. Thus, the initial validity of "LISTSET(sp,n,next) U LISTSET(p,n,next) = $(\text{LISTSET}(sp_o,n,next))_o$" directly implies the final validity of "LISTSET(sp,n,next) = $(\text{LISTSET}(sp_o,n,next))_o$".

Two loops must be tested for termination. The inner loop terminates because line (17) assigns to np the old value of np.next, and "LIST(sp,next)" and "sp$\Rightarrow$np" are invariant within the loop, so "LIST(np,next)" is true within the loop, and the conditions for Lemma L.24 are met. For the outer loop, each iteration assigns to p the old value of p.next, either on line (9) or line (20), and again "LIST(p,next)" is true throughout. Thus, Lemma L.24 can again be used to prove its termination. Since both loops terminate, the program is guaranteed to terminate.

Chapter 5

CONCLUSIONS

A. Summary

The process we went through in Chapter 4 is a good example application of the approach described in Chapter 3. The first step was to isolate a specific class of data structures—in this case, singly-linked lists—and to define the class precisely in an invariant. From the definition, we were able to immediately deduce some useful corollaries. We then defined some representation functions, mapping elements of the structural class to sets and sequences in a fairly obvious way. The function definitions led to several more potentially useful corollary results. We had now arrived at a point where we could analyze how various types of source-language statements might affect the validity of the class invariant and the values of the representation functions. These results were embodied in a series of lemmas, along with an important result concerning loop termination. With this logical foundation, we were then able to prove the correctness of three programs which operate on structures from the class. In doing so, we realized the need for one more type of lemma—one which covers all those cases which have no effect at all on the invariant and represen-

tation functions.

The two people whose work has most influenced my approach are Hoare and Burstall. Hoare has done so much in the area of verification that it is doubtful whether a work such as mine could exist at all without him. He was the first to systematically analyze the semantics of source languages with relation to verification [Hoa69]. His axioms are both the historical predecessors of my verification lemmas and the logical foundation upon which the lemmas rest (and by which they can be proved, as in Appendix B). The concepts of the "invariant" and the "representation function" both originated with him ([Fol71] and [Hoa72b] respectively). His axiomatizations of abstract data classes [Hoa72a] are an important basis for the use of representation functions.

Burstall's work on LISP-type lists [Bur72a] is the single closest approach to mine. Working with a specific subclass of Lists, Burstall introduced a notation for "connection" somewhat similar to what I use, and stated a series of axioms analagous to my "verification lemmas". His concepts were narrower in scope than mine, since his domain of interest was more restricted, but it is not hard to see how they could be generalized to apply to larger classes of structures.

Having acknowledged these debts, I still feel that the approach presented in this thesis represents a new and valuable method for handling data structures in proving the correctness of programs. The major benefit of the work is in the area of data characterization rather than proof technique itself. That is, while the verification lemmas help ease the task of actually proving the verification conditions, they are not essential. The same programs could be proved correct without them, using basic axioms like those of Hoare and by resorting directly to the definitions of the invariant and the representation functions. While the proofs would be considerably more difficult and detailed, they would still be manageable.

Without the descriptive mechanisms I have introduced, however, it is difficult to see how the assertions for these programs could be expressed. Consider, for instance, the third example program of Chapter 4, which performs an insertion sort on a list. I know of no suitably formal method for stating the output condition of this program—that the output list is a sorted version of the input list—without using notation such as mine. The intermediate assertions would be even more difficult to express.

What has held back work on verifying programs which operate on data structures up to now has been a lack of tools for describing structures. Because a structure is so much bigger and more complex than a single variable, and because its exact size and form may not be precisely known to a program which operates on it, new problems arise in terms of characterizing it. In particular, characterizing structures requires two capabilities additional to those used for elementary data. Unlike the simple variable, which is regarded as unitary and undecomposable, the data structure must in part be viewed in terms of its internal structure. On the other hand, a structure must sometimes be considered not only as a single object, but as merely the concrete embodiment of a data abstraction. By recognizing these two almost opposite needs, and combining them together with the practical requirements of correctness proofs, my approach represents an important step toward a total system for handling data structures in proofs of correctness.

B. Relation to structured programming

Dijkstra [Dij72] and Hoare [Hoa72a] have been among the leading proponents of "structured programming" as a design tool for writing programs. Under the regimen of structured programming, the programmer first constructs

his program at a very high level of abstraction; i.e., operations and data are expressed only as abstract concepts. The program is then refined by successively specifying these concepts in terms of lower-level constructs, some of which may themselves be abstractions which must subsequently be specified. The process ends when all constructs in the program are expressed in terms of features in the source language.

Both Dijkstra and Hoare advocate the practice of creating a program and its proof simultaneously, arguing that each task is made easier by performing it along with the other. My approach to proving correctness is particularly well-suited to use in conjunction with structured programming. My two major concepts of "invariants" and "representation functions" both correspond to important ideas in structured programming. The invariant of a structure is the lower-level specification of a complex abstract data object. The representation function is the mapping from the object at a lower level of abstraction to the abstract object at a higher level which it represents.

Thus, for instance, the assertions of a program can be constructed at a point where the program's data and the basic operations on that data are specified as

abstractions. The actual abstraction can then be formed by substituting for each abstract data object the representation function mapping the data structure in the source-language program into that abstraction; the specification of each structure at the lower level is then added as its invariant to complete the assertion. This simplifies the task of constructing assertions, since at the higher level of abstraction the operation of the program is clearer, and thus the assertions are easier to create.

C. Further research

The approach toward handling data structures which this thesis describes and illustrates is intended to be of general applicability. Its use should not be limited to data structures similar to lists, nor to source languages resembling the one used here. Yet much of this is pure speculation, as I have had a chance to try out the approach on a few classes of structures only.

A more complex class of structures than lists will require greater complexity in the class invariant and the representation functions, and consequently a larger number of corollaries and lemmas to create a really effective formalism. In particular, the greater the number of reference components per record, and the more

invariant relations between these components, the greater will be the complexity required. (In fact, these two factors could serve as reliable metrics for the complexity of a structural class.) The class of "family trees" introduced in Chapter 2 and defined (in an invariant) in Chapter 3, for instance, requires a quite extensive treatment. Whether the concepts I introduce here are adequate for even more complex structural classes is an issue to be examined.

The situation is similar with respect to other source languages. As noted in Chapter 2, the language used here was designed to incorporate a good deal of static type- and component-checking. Use of a language like PL/I, in which such checking is considerably weaker, would create a greater burden on the assertions and on the proofs. Whether an approach such as mine is adequate for such a language remains to be seen. It seems that several new primitives would be required at least, to characterize such properties as "component-existence." A language like LISP has its own special aspects, of course, which would have to be handled specially.

As this discussion indicates, the example of Chapter 4 represents something like the minimum in inherent complexity for an application of my approach. Use of

other source languages and less simple structural classes requires that much more detail. This complexity can be limited, however.

A variation which I feel has strong possibilities is to generalize one full level. Instead of proving lemmas about a single invariant, it should be possible to prove "meta-lemmas" which are valid for any invariant in a broad class. That is, the validity of a certain kind of lemma may hinge on one or two particular properties of the class invariant; any invariant for which those properties hold will admit such a lemma.

The lemmas in Chapter 4 for lists, for example, could easily be generalized to cover any structural class (such as binary trees) which is recursively defined and which prohibits shared substructures. Embodying such results in meta-lemmas would make them available for any structural class so defined.

Ideas such as this can carry the approach of this thesis forward to the point where any program can be quickly and precisely verified. Perhaps then, verification will become a routine part of program construction; error-free programs will finally then become the norm.

BIBLIOGRAPHY

[Ash71] Ashcroft, R. and Z. Manna, "The translation of 'goto' programs to 'while' programs," Information Processing 71, Vol. I, C. V. Freiman (ed.), North-Holland Publishing Co., Amsterdam, 1972.

[Bur69] Burstall, R. M., "Proving properties of programs by structural induction," Computer Journal, 12,1,41-68 (February 1969).

[Bur72a] Burstall, R. M., "Some Techniques for Proving Correctness of Programs which Alter Data Structures," Machine Intelligence 7, D. Michie (ed.), American Elsevier, New York, 1972.

[Bur72b] Burstall, R. M. and R. Topor, Mechanizing Program Correctness by Symbolic Interpretation, (work in progress), Department of Machine Intelligence and Perception, University of Edinburgh, November 1972.

[Cli72] Clint, M. and C. A. R. Hoare, "Program Proving: Jumps and Functions," Acta Informatica, 1, 214-224 (1972).

[Deu73] Deutsch, L. P., An Interactive Program Verifier, Report No. CSL-73-1, Xerox Corp., Palo Alto, May 1973.

[Dij68] Dijkstra, E. W., "Goto statements considered harnful," Communications ACM, 11, 3, 147-8 (March 1968).

[Dij72] Dijkstra, E. W., "Notes on Structured Programming," Structured Programming, Academic Press, New York, 1972.

[Els72] Elspas, B., K. M. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness," Computing Surveys, 4, 2, 97-147 (June 1972).

[Flo67] Floyd, R. W., "Assigning meanings to programs," Proceedings of a Symposium in Applied Mathematics, American Mathematical Society, J. T. Schwartz (ed.), Providence, 1967.

[Fol71] Foley, M. and C. A. R. Hoare, "Proof of a recursive program: Quicksort," Computer Journal, 14, 4, 391-5 (November 1971).

[Gol63] Goldstine, H. R. and J. von Neumann, "Planning and coding problems for an electronic computer instrument, part 2, vol. 1-3," John von Neumann collected works, Vol. 5, A. H. Traub (ed.), Pergamon Press, New York, 1963.

[Goo70] Good, D. I., Toward a Man-Machine System for Proving Program Correctness, University of Texas, TSN-11, Austin, June 1970.

[Gre72] Greif, I. G., Induction in Proofs About Programs, Project MAC, M.I.T., MAC-TR-93, Cambridge, February 1972.

[Hoa68] Hoare, C. A. R., "Record Handling," Programming Languages, F. Genuys (ed.), Academic Press, New York, 1968.

[Hoa69] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," Communications ACM, 12, 10, 576-583 (October 1969).

[Hoa71] Hoare, C. A. R., "Procedures and Parameters: An Axiomatic Approach," Symposium on Semantics of Algorithmic Languages, E. Engeler (ed.), Springer Verlag, New York, 1971.

[Hoa72a] Hoare, C. A. R., "Notes on Data Structuring," Structured Programming, Academic Press, New York, 1972.

[Hoa72b] Hoare, C. A. R., "Proof of Correctness of Data Representations," Acta Informatica, 1, 271-281 (1972).

[Kin69] King, J. C., A Program Verifier, Computer Science Department, Carnegie-Mellon University, Pittsburgh, September 1969.

[Kin71] King, J. C., "A program verifier," Information Processing 71, Vol. I, C. V. Freiman (ed.), North-Holland Publishing Co., Amsterdam, 1972.

[Knu68] Knuth, D. E., The Art of Computer Programming, Vol. I, Addison-Wesley, Reading, 1968.

[Man69] Manna, Z., "The Correctness of Programs," Journal of Computer and System Sciences, 3, 2, 119-127 (May 1969).

[Man71] Manna, Z. and R. J. Waldinger, "Toward Automatic Program Synthesis," Communications ACM, 14, 3, 151-165 (March 1971).

[McC63] McCarthy, J., "A Basis for a Mathematical Theory of Computation," Computer Programming and Formal Systems, North-Holland Publishing Co., Amsterdam, 1963.

[Mor72] Morris, J. H., Verification-oriented Language Design, University of California-Berkeley Computer Science Technical Report No. 7, Berkeley, December 1972.

[Nau66] Naur, P., "Proofs of algorithms by general snapshots," BIT, 6, 4, 310-316 (1966).

[Pou72] Poupon, J. and B. Wegbreit, Covering Functions, Center for Research in Computing Technology, Harvard University, Cambridge, September 1972.

[Sta67] Standish, T. A., A Data Definition Facility for Programming Languages, Carnegie-Mellon University, Pittsburgh, May 1967.

[Wir66] Wirth, N. and C. A. R. Hoare, "A Contribution to the Development of ALGOL," Communications ACM, 9, 6, 413-429 (June 1966).

[Woz71] Wozencraft, J. M. and A. Evans, Notes on Programming Linguistics, Department of Electrical Engineering, M. I. T., 1971.

APPENDIX A

AXIOMATIZATION OF SEQUENCE OPERATIONS

Hoare's paper "Notes on Data Structuring"[Hoa72a] describes and axiomatizes several abstract data classes. Since representation functions have been defined in the thesis to yield only sets and sequences, we need axiomatize only these two classes. Set theory is well enough established that its results need not be explicitly stated but instead can simply be referred to as needed. Since this is not the case for sequences, however, this axiomatization is needed.

The basic operation on sequences is concatenation, which is performed by the binary prefix operator "concat". It is taken as basic, and is not axiomatized beyond the fact that it is associative. The basic operators for breaking down a sequence into its component parts are the unary prefix operators "first" and "last", which yield the first and last items of a non-empty sequence, respectively, and "initial" and "final", which remove the last or first item of a non-empty sequence, respectively. The domain of sequences is defined in terms of an element domain D, with representative member d. As noted previously, "[ ]" signifies the empty sequence, and " [d] " signifies the sequence with single element d.

The class of sequences S with elements from domain D is defined:

(1) [] is an element of S.

(2) If s is an element of S and d is an element of D, then concat(s, [d]) is an element of S.

(3) The only elements of S are as specified in (1) and (2) above.

The operations on sequences can then be axiomatized:

(4) concat(x,concat(y,z)) = concat(concat(x,y),z)).

(5) last(concat(s, [d])) = d.

(6) initial(concat(s, [d])) = s.

(7) first([d]) = d.

(8) $s \neq [\,] \supset$ first(concat(s, [d])) = first(s).

(9) final([d]) = [].

(10) $s \neq [\,] \supset$ final(concat(s, [d])) = concat(final(s), [d])).

(11) last, initial, first, and final are not defined for [].

APPENDIX B

PROOF OF LEMMA L.19

Lemma L.19 in Chapter 4 expresses the result of a statement in the source language which updates the value of a link component in a record which is part of a list. That is, it states the result of the statement "q.next := expr", where "expr" is any expression other than a record-creation expression, given the precondition that "LIST(p,next) and ($p \overset{next}{\Longrightarrow} q \overset{next}{\Longrightarrow} +$ null)" is true. The result, in words, is that "$p \overset{next}{\Longrightarrow} q$" is still true; that for any r, $q \overset{next}{\Longrightarrow} +$ r if and only if expr $\overset{next}{\Longrightarrow}$ r; that p still references a list if and only if expr references a list not containing q; and that if p still references a list, it is the adjoining of the original sublist from p to q inclusive, with the list referenced by expr, so that the values of the representation functions are the union and concatenation, respectively, of the functions for these two sublists.

This result can be proved formally using the axioms and rules of inference of Hoare [Hoa69]. What will be proved here is actually just a part of the lemma; the conditions under which "LIST(p,next)" is claimed to be preserved will be assumed true, and the result will then be proved for that case. To complete the proof of

the lemma, it would be necessary to show that if either "LIST(expr,next)" were false or "expr $\overset{next}{\Longrightarrow}$ q" were true, then "LIST(p,next)" is not preserved. (It would also be necessary to show that the other clauses in the postcondition still followed, but these are all either totally independent of the "LIST(p,next)" result, such as that "p $\overset{next}{\Longrightarrow}$ q" is preserved; or else trivial without the truth of "LIST(p,next)"—the clause for the values of the representation functions is true if "LIST(p,next)" is false because the left-hand-side of the implication then becomes false,making the implication true.) Proving that "LIST(p,next)" is not preserved under either of these conditions is quite uninteresting and totally obvious. Proving the case for which "LIST(p,next)" is preserved, however, is interesting, since it shows how general axioms like those of Hoare can be used to prove the more specific and more immediately useful result embodied in a verification lemma.

The result to be proved, then, is " P {Q} R " , where the precondition P is the following:

$$\lceil \text{LIST(p,next)} \ \underline{\text{and}} \ (p \overset{next}{\Longrightarrow} q \overset{next}{\Longrightarrow} + \ \underline{\text{null}}) \ \underline{\text{and}}$$
$$\text{LIST(expr,next)} \ \underline{\text{and}} \ \underline{\text{not}}(\text{expr} \overset{next}{\Longrightarrow} q) \rceil \quad ,$$

the code Q is the statement "q.next := expr", and the postcondition R is the following:

$$\Big[\text{LIST}(p,\text{next}) \ \underline{\text{and}} \ (p \overset{\text{next}}{\Longrightarrow} q) \ \underline{\text{and}}$$
$$(q \overset{\text{next}}{\Longrightarrow}{}^{+}\ r \quad \underline{\text{iff}} \quad \text{expr} \overset{\text{next}}{\Longrightarrow} r) \ \underline{\text{and}}$$
$$(q \overset{\text{next}}{\Longrightarrow}{}^{+}\ r \supset$$

$$\text{SUBLISTSET}(p,r,\text{sel},\text{next}) =$$
$$(\text{SUBLISTSET}(p,q.\text{next},\text{sel},\text{next}))_1 \ \cup$$
$$(\text{SUBLISTSET}(\text{expr},r,\text{sel},\text{next}))_1 \quad \underline{\text{and}}$$

$$\text{SUBLISTSEQ}(p,r,\text{sel},\text{next}) =$$
$$\text{concat}((\text{SUBLISTSEQ}(p,q.\text{next},\text{sel},\text{next}))_1,$$
$$(\text{SUBLISTSEQ}(\text{expr},r,\text{sel},\text{next}))_1))\Big] \ .$$

By Hoare's Rules of Consequence, " $P \{Q\} R$" can be proved by proving some " $S \{Q\} T$" such that $P \supset S$ and $T \supset R$. Furthermore, Hoare's Axiom of Assignment guarantees the validity of " $A^{q.\text{next}}_{\text{expr}} \{Q\} A$" for any assertion A, where A^{x}_{e} represents the result of substituting e for all free occurrences of x in A. (Intuitively, what this axiom says is that any assertion which can be made about "expr" before the assignment can be made about "q.next" after the assignment.)

So, the proof can be accomplished by finding a T such that $T \supset R$ and $P \supset T^{q.\text{next}}_{\text{expr}}$, since $T^{q.\text{next}}_{\text{expr}}$ will serve as an acceptable S for use with the Rules of Consequence. Let T be the following:

$$\lceil \text{LIST}(p,\text{next}) \ \underline{\text{and}} \ (p \overset{\text{next}}{\Longrightarrow} q \overset{\text{next}}{\Longrightarrow}+ \ \underline{\text{null}}) \ \underline{\text{and}}$$

$$(q.\text{next} \overset{\text{next}}{\Longrightarrow} r \ \ \underline{\text{iff}} \ \ \text{expr} \overset{\text{next}}{\Longrightarrow} r) \ \ \underline{\text{and}}$$

$$\text{LIST}(q.\text{next},\text{next}) \ \underline{\text{and}} \ \ \underline{\text{not}}(q.\text{next} \overset{\text{next}}{\Longrightarrow} q) \ \underline{\text{and}}$$

$$(q \overset{\text{next}}{\Longrightarrow}+ \ r \ \supset$$

$$\text{SUBLISTSET}(p,q.\text{next},\text{sel},\text{next}) =$$

$$(\text{SUBLISTSET}(p,q.\text{next},\text{sel},\text{next}))_1$$

$$\underline{\text{and}} \ \ \text{SUBLISTSET}(q.\text{next},r,\text{sel},\text{next}) =$$

$$(\text{SUBLISTSET}(\text{expr},r,\text{sel},\text{next}))_1$$

$$\underline{\text{and}} \ \ \text{SUBLISTSEQ}(p,q.\text{next},\text{sel},\text{next}) =$$

$$(\text{SUBLISTSEQ}(p,q.\text{next},\text{sel},\text{next}))_1$$

$$\underline{\text{and}} \ \ \text{SUBLISTSEQ}(q.\text{next},r,\text{sel},\text{next}) =$$

$$(\text{SUBLISTSEQ}(\text{expr},r,\text{sel},\text{next}))_1)\rceil \ .$$

There are now two things to be shown. First, that $T \supset R$. And second, that $P \supset T^{q.\text{next}}_{\text{expr}}$.

To show that $T \supset R$: R consists of four clauses, so we must show that each of these clauses follows from T.

"LIST(p,next)" and "$(p \overset{\text{next}}{\Longrightarrow} q)$" both follow directly from the fact that each is included within T.

"$(q \overset{\text{next}}{\Longrightarrow}+ \ r \ \ \underline{\text{iff}} \ \ \text{expr} \overset{\text{next}}{\Longrightarrow} r)$" follows from the clause "$(q.\text{next} \overset{\text{next}}{\Longrightarrow} r \ \ \underline{\text{iff}} \ \ \text{expr} \overset{\text{next}}{\Longrightarrow} r)$" in T. In fact, these two clauses are equivalent, a fact which follows directly from the definitions of "$\Longrightarrow$" and "$\Longrightarrow+$".

The final clause in R is the implication dealing with the representation functions, beginning

"$q \overset{next}{\Longrightarrow}+ r \supset \ldots$" Since "$(p \overset{next}{\Longrightarrow} q)$" is in both R and T, Corollary L.12 can be used to show that

(*) SUBLISTSET(p,r,sel,next) =

SUBLISTSET(p,q.next,sel,next) U

SUBLISTSET(q.next,r,sel,next) and

SUBLISTSEQ(p,r,sel,next) =

concat(SUBLISTSEQ(p,q.next,sel,next),

SUBLISTSEQ(q.next,r,sel,next)),

given the assumption that $q \overset{next}{\Longrightarrow}+ r$. Then the presence of the two clauses

"SUBLISTSET(p,q.next,sel,next) = $(\text{SUBLISTSET}(p,q.\text{next}_i,$ sel,next$))_i$"

and "SUBLISTSET(q.next,r,sel,next) =

$(\text{SUBLISTSET}(\text{expr},r,\text{sel},\text{next}))_i$"

on the right-hand-side of the implication in T justifies their substitution (on the right-hand-side of the implication in R) into statement (*) above to yield

"SUBLISTSET(p,r,sel,next) =

$(\text{SUBLISTSET}(p,q.\text{next},\text{sel},\text{next}))_i$ U

$(\text{SUBLISTSET}(\text{expr},r,\text{sel},\text{next}))_i$"

on the right-hand-side of the implication. A similar argument can be made for the part of the clause concerning SUBLISTSEQ.

Thus, T $\supset$ R is established.

Now consider $T_{expr}^{q.next}$. Making the substitution, we get:

$$\begin{aligned}
&[\text{LIST}(p,\text{next}) \ \underline{\text{and}} \ (p \overset{\text{next}}{\Longrightarrow} q \overset{\text{next}}{\Longrightarrow} + \ \underline{\text{null}}) \ \underline{\text{and}} \\
&(\text{expr} \overset{\text{next}}{\Longrightarrow} r \ \ \underline{\text{iff}} \ \ \text{expr} \overset{\text{next}}{\Longrightarrow} r) \ \ \underline{\text{and}} \\
&\text{LIST}(\text{expr},\text{next}) \ \ \underline{\text{and}} \ \ \underline{\text{not}}(\text{expr} \overset{\text{next}}{\Longrightarrow} q) \ \ \underline{\text{and}} \\
&(q \overset{\text{next}}{\Longrightarrow} + \ r \supset \\
&\quad \text{SUBLISTSET}(p,\text{expr},\text{sel},\text{next}) = \\
&\qquad (\text{SUBLISTSET}(p,\text{expr},\text{sel},\text{next}))_1 \ \ \underline{\text{and}} \\
&\quad \text{SUBLISTSET}(\text{expr},r,\text{sel},\text{next}) = \\
&\qquad (\text{SUBLISTSET}(\text{expr},r,\text{sel},\text{next}))_1 \ \ \underline{\text{and}} \\
&\quad \text{SUBLISTSEQ}(p,\text{expr},\text{sel},\text{next}) = \\
&\qquad (\text{SUBLISTSEQ}(p,\text{expr},\text{sel},\text{next}))_1 \ \ \underline{\text{and}} \\
&\quad \text{SUBLISTSEQ}(\text{expr},r,\text{sel},\text{next}) = \\
&\qquad (\text{SUBLISTSEQ}(\text{expr},r,\text{sel},\text{next}))_1)] \ .
\end{aligned}$$

Each of the four equalities on the right-hand-side of the implication is true trivially, since in the precondition, all values are by definition the initial values, so the whole implication is trivially true. Similarly, the clause "$(\text{expr} \overset{\text{next}}{\Longrightarrow} r \ \ \underline{\text{iff}} \ \ \text{expr} \overset{\text{next}}{\Longrightarrow} r)$" is trivially true. The other four clauses are in fact the four clauses of P, so this establishes the fact that $P \supset T_{expr}^{q.next}$. By the reasoning given previously, this completes the proof.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 2/29/96

Report # LCS-TR-124

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 144 (149-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☒ Other: BIBLIOGRAPHIC DATA SHEET

Page Data:

Blank Pages(by page number): FOLLOWS LAST PAGE (143)

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP :	(1-144)
UNT'ED TITLE PAGE,	2-143,
UNT'ED BLANK	
SCANCONTROL, BIBLIOGRAPHIC DATA SHEET, TRGT'S	(145-149)
	(3)

Scanning Agent Signoff:

Date Received: 2/29/96 Date Scanned: 3/1/96 Date Returned: 3/1/96

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**