PROGRAM VERIFICATION IN FUNCTIONAL PROGRAMMING SYSTEMS

by

James Leland Silver, Jr.

Thesis submitted to the Graduate Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

APPROVED:

D. J. Martin, Chairman

T. E. Lindquist

D. C. Allison

May, 1983

Blacksburg, Virginia

Acknowledgments

I wish to thank Dr. Johannes J. Martin for his very kind assistance during the preparation and submission of this thesis. His suggestions and guidance were instrumental in assuring the ultimate success of this enterprise.

I would also like to acknowledge my debt to the members of the Computer Science and Applications Department for shaping the way in which I view computing as an academic discipline.

Table of Contents

Introduction

The problem of demonstrating that a program performs as required has a history as long as that of programming itself and, although the computing community has relied and continues to rely largely upon testing the program under simulated conditions in its attempts to provide such demonstrations, since the publication of Floyd's paper, [5], in 1967, there has never been a lack of efforts to prove mathematically that programs satisfy a particular set of design specifications. This is a part of a more general effort to develop well founded programming systems which are sufficiently powerful and eloquent to allow for the statement and proof of theorems about the programs in such a system.

Of course, if the sole concern were to prove that a program satisfies a particular theorem, then our efforts might be concentrated upon developing automatic verification methods. Our interest in manual verification methods would then be limited to providing a supplement to be used when automatic verification fails and to providing insights into the development of automatic methods. This is not, however, the case. The collection of techniques which are generally regarded as manual program verification also provide a number of additional benefits. They provide a means for

specifying the semantics of a programming system. This may be in terms of a set of axioms which describe the behaviour of the primitives of the system in a purely formal way or a set of previously proven theorems which guarantee that a particular implementation satisfies its axiomatic description. These techniques provide a deeper insight into the programs themselves and yield guidelines for the design of such programs. Indeed, as Dijkstra observes in [4], it is possible to begin with the statement of the desired theorem and design the program and the proof of the theorm simultaneously so that the program satisfies any assumptions required for the proof. When applied to programs which are incorrect, these methods often suggest the appropriate correction. Finally, such methods provide a deeper understanding of the problems of program specification in that they require a clear, concise statement of what the program is expected to do.

There are a number of factors which determine the success and utility of verification methods. Of these, several seem to be pre-eminent. Their success certainly depends to a large degree upon the ease with which program specifications may be written and the extent to which these specifications deal with the same objects that the program will deal with as opposed to using artificially contrived mechanisms to describe aspects of the computation which are

external to the program itself. Their success also depends upon the existence of a formal semantic definition of the programming system being used. Such a definition is necessary as a foundation for correctness proofs. It is virtually impossible to provide useful verification methods for systems which rely on implementation dependent definitions of their semantics. They depend upon the accessibility and applicability of a body of general theorems about programming so that each new system does not have to be entirely re-invented. Finally, they depend upon the ability to separate the program into units which interact in limited ways. Without the ability to view programs as a collection of essentially independent components, manual verification of any but the most trivial programs would be impossible.

In functional programming systems, both the programs and the theorems about the programs deal directly with values. In this sense, there is a much closer relationship between the programs and the specifications than is possible with Algol-type languages. Indeed, it is often possible to frame a theorem about the correctness of a program in a functional programming system as a predicate function in the language in which the program is written. This is in contrast to traditional Algol-type languages in which the programs manipulate storage locations while the specifications are

stated in terms of values. It is the problem of providing a mechanism with which to represent the current state of a mutable store which has been one of the chief obstacles to program verification in traditional programming systems. Functional programming systems require no such contrivances for the representation of state transformations during a computation. This natural relationship between the programs and program specifications in a functional programming system extends to the semantic description of the system itself. It provides the designers of such systems a vehicle for clear, concise descriptions of their systems; it allows the proposed systems to be easily checked for consistency and completeness; it permits simple and virtually foolproof validation of implementations of such a system; and it provides the theoretical foundation for correctness proofs.

Functional programming systems exhibit a close relation to the general body of results from computability theory making these results more readily available than is possible in traditional programming languages. Results involving inductive properties of programs and least fixed points are usually phrased in terms of functions and must be reworked before they can be applied to program verification in Algol-type languages [10,210] and [8,528]. When applied to functional programming systems, no such changes are necessary.

Finally, and perhaps most importantly, it is the complete absence of side effects of components of a larger functional program which makes functional programming systems so amenable to verification methods. It is never necessary to guard against improper interaction between components of a functional program since such interaction is impossible. Once the individual components of such a program have been shown to function correctly, the propositions which describe their behaviour may be accepted as axioms in the verification for the higher level program.

This paper will present several examples of programs written in a functional programming system together with theorems describing their behaviour and proofs of those theorems. The purpose is to demonstrate the ability of functional programming systems to provide a natural representation of a program, and its verification. In so doing, we will observe the ease with which existing results in the theory of computation may be used.

The paper begins with a brief survey of the history of manual verification methods, citing some of the more important contributions in this area. These include Floyd's inductive assertion method, Hoare's axiomatic method, and the constructive approach of Dijkstra's weakest precondition methods. In the area of proving termination of iterative and recursive programs, we cite the fixed point theories

developed by Scott, Manna and others and several inductive approaches. Chapter 2 contains a description of the functional programming system which will be used in this paper together with those theorems which will form the theoretical basis for the verification of programs in this system. The system to be used is the programming language FAD described in [9]. This language includes several features designed to make programs more readable. These include the use of infix operators and local names for selector functions. These features will be used freely in the material which follows since they do make both the programs and the accompanying theorems and their proofs easier to follow, but it should be emphasized that the objective of this paper is not to provide an apologia for a particular functional programming system, but to to demonstrate the utility of any such system as a vehicle for representing verifiable programs.

Chapter 3 presents several sample FAD programs together with theorems which describe the behavior of the programs and verifications for the theorems. It is characteristic of such systems that these theorems are stated as predicates in the system being used together with universal and existential quantifiers. The proof techniques employed rely on the fixed point theorems to prove termination and an axiomatic description of the functional programming system to prove

correctness.

The last chapter contains a review of the results of Chapter 3 and, in order to demonstrate the amenability of functional programming systems to other methods, describes a simple application of Dijkstra's weakest preconditions to derive and verify a program in a functional programming system.

1 Literature Review

1.1 Introduction

We shall begin by examining several of the mainstream verification methods which have been developed since the late sixties. In general, the developers of these methods have tended to separate the problem of showing that a program terminates from that of showing that a properly terminating program returns the desired results. A program which has been shown to satisfy the second condition is said to be partially correct, while a partially correct program which has been proven to terminate is totally correct. Methods which are intended primarily to show that a terminating program returns the correct results may generally be classified as deductive or constructive. The deductive methods are intended to be applied to an existing program to show that it is valid while the constructive methods attempt to produce verified programs from problem specifications.

1.2 Inductive Assertion Method

The earliest of the deductive methods is the inductive assertion method which was introduced by Floyd in [5]. This method requires the user to trace the program, providing an

assertion for each of the edges of each possible path through the program. Then the strongest verifiable consequent (svc) predicate transform for the language being used is applied to the disjunction of all the assertions on the incoming edges at each syntactic unit of the program. The result should be sufficient to guarantee the assertion on each of the outgoing edges. If this process is successful at each of the syntactic units in the program and the input specifications are used as the assertion for the start of the program then one may conclude that the output specification is the disjunction of the assertions on all terminal paths of the program. Thus, if the assertions have been well chosen, one may demonstrate that, assuming termination, the program is correct.

The inductive assertion method enjoys the benefit of being fairly automatic once the assertions have been determined. However, it has several shortcomings. Improper choice of assertions may result in the failure to verify a program which is in fact correct and, in the case of an incorrect program, a failure of the verification method does not necessarily indicate an appropriate correction. Finally, the semantic description of the programming system being used is hidden in the definitions for the svc predicate transform.

1.3 Axiomatic Method

In [6], C. A. R. Hoare proposed a modification of the inductive assertion method which has come to be known as the axiomatic method. This method replaced the strongest verifiable consequent predicate transform with a set of axioms and rules of inference. It also replaced the flowcharts used in Floyd's original work with program text and it relinquished some of the more automatic features of the inductive assertion method. In fact, Hoare's objective was to do more than provide a mechanism for verifying programs. Rather, it was to "elucidate the axioms and rules of inference which underlie our reasoning about computer programs." [6,576] These rules and axioms formed the basis of a deductive system similar to Floyd's in which it is possible to formulate and prove theorems describing the behavior of programs or parts of programs.

These modifications provide several significant advantages. The semantic description of the programming language being used is readily apparent in the axioms and rules of inference presented. Further, the verification of the program is usually subdivided into natural semantic components as opposed to the rather arbitrary subdivision into syntactic units characteristic of the inductive assertion method. Thus, failure to verify an incorrect

program is more likely to lead the programmer to a correction in the axiomatic approach than with inductive assertion.

1.4 Weakest Preconditions

Although either of the above methods may be employed to guide the design of a program, they are primarily applied to existing programs. An alternative method is the constructive approach proposed by Dijkstra in [4]. In general terms, the process involves determining the desired postcondition for a program and, by applying the weakest precondition predicate transformer to this predicate and various syntactic units, to choose a syntactic unit which provides a precondition which is in some sense closer to the precondition for the entire program. This process is repeated until a precondition is achieved which is a direct consequence of the input specifications. Thus the design process is guided by both the programmer's experience and intuition and the information provided by the weakest precondition predicate transformer. The result is a completed program together with its verification.

1.5 Inductive Methods

Dijkstra's weakest precondition method proves total correctness of the program constructed and both the inductive assertion method of Floyd and Hoare's axiomatic approach have extensions which prove termination. Nevertheless, the main thrust of these methods is to prove that the program under investigation returns the correct results. There are, however, other methods available which concentrate their efforts on proving termination. Indeed, it is possible to apply these methods to a recursive function, for example, and prove that the function will terminate without being able to give the value returned by the function in a closed form. Nevertheless, there are a number of techniques available for proving theorems about the behavior of such functions. When such methods are applied to iterative or recursive programs, they are bound by the nature of those programs to be inductive. These methods are usually classified by the object of the induction being performed as computational induction methods and structural induction methods.

The computational induction methods are generally applied to recursive programs and are so named because the object of the induction is the depth of the recursion. These methods are founded on the theory of least fixed points of

recursive functions developed by Kleene. This theory guarantees that, assuming an appropriate computation rule, the function defined by any one of a large class recursive programs can be described as the limit of a sequence of partial functions [8,529]. This is sufficient to ensure the termination of the program when the input values belong to the domain of the function so described. The basic theorem of computational induction observes that, for certain admissible predicates, it is possible to prove that the limit function satisfies the predicate by proving inductively that all of the partial functions in the defining sequence satisfy that property [8,530]. (The admissible predicates are precisely those predicates whose satisfaction by each of the functions in the sequence is sufficient to guarantee their satisfaction by the limit function.) The methods known as structural induction use the data structures of a computation as the object of their induction. The various methods actually consist of extensions of the principles of mathematical induction for well-ordered sets to more general classes of partially ordered sets. These include well-founded sets (partially ordered sets which contain no infinite decreasing sequence) [2] and, more recently, multiset orderings [3].

In [1], Backus notes the many advantages of functional programming systems. In particular, he cites the ability to

develop an algebra of programs from a set of axioms describing such a system and the close relationship between functional programming systems and the theoretical foundations of program verification.

1.6 Summary

In the sections which follow, we present a description of the functional programming system being used which may be taken as axiomatic. This allows us to draw a number of conclusions about the properties of the example programs we shall consider in the style of Hoare's axiomatic method. However, we shall rely upon the theory of fixed points in the form presented in [1] to prove termination of these programs. Although there are no data structures in the usual sense, these results also provide descriptions of the structure of the intermediate values to which inductive arguments will be applied to demonstrate the correctness of these programs. Thus, in some sense, these proofs will be equivalent to structural induction arguments although the full generality of structural induction will not be required.

2 Fundamental Theories

2.1 Preliminaries

In order to make this paper reasonably self-contained, we will begin with a brief description of functional programming system FAD. We shall include below only those features which are pertinent to our discussions and shall rely on informal definitions whenever possible. For a complete, formal description of the system, see [9].

Similar to other functional programming systems, FAD begins with a set of atoms which includes numbers, character strings, the boolean values T and F, and the empty sequence, NIL. These atoms are used to construct items which are recursively defined as being either atoms or sequences of items between angular brackets. Thus <3,<4,NIL>,5> is an item.

The items are in turn used to construct the basic domain of FAD, a set consisting of formal expressions which denote all finite sets of items and certain infinite sets of items. These expressions include all of the items; the meta-symbol PHI, which represents the empty set; and any expression of the form <A>, A:B, or <A,C> where A, B, and <C> are themselves item expressions. In its usage, PHI corresponds intuitively to 'undefined'.

For our purposes, we may restrict our attention to the set of items which we will denote ITEM. We will, however, make use of the union operation (':') and the sequencing operation in order to describe subsets of ITEM. Thus <A,C> = { <a,c> : a in A and <c> in <C>}. We will often use these notations to specify recursively defined infinite sets. These sets will be used to describe the domain and range of the functions considered. We will also make free use of the symbolism, ==, for associating a name with a set of items. Thus, for example, the set of all sequences whose entries are integers may be described by SEQ == NIL:<integer,SEQ>.

Similarly, we may restrict our attention to functions which map some subset of the set ITEM into ITEM. Given any such function, say f, we denote the result of applying f to an item x by f:x. If x is an item which is not in the domain of f, then we may extend the domain of f to include x by defining f:x to be the meta-symbol PHI. Since PHI is not itself an item, we set ITEM+ = ITEM U {PHI}. We may now view all partial functions defined on ITEM as functions from ITEM+ to ITEM+. In this sense, the domain of each of the functions we will use is ITEM+. We shall refer to the original domain of such an extended function as its restricted domain. Thus the restricted domain of a function f is {x in ITEM : f:x <> PHI}.

The functions we will consider will be built up from

primitive functions using functional forms. We list below those primitive FAD functions which appear in this paper. In order that we may apply the theory of fixed points developed in [7], we need to exercise particular care in observing when a function returns PHI as its value. In the following definitions, X, Y, and Z represent items and W represents a string of symbols such that <W> is an item. If the string W is empty, then <W> is understood to represent NIL. In each definition, the conventions regarding these symbols implicitly designate a restricted domain. If the function is applied to any element not in that restricted domain, the element PHI will be returned.

2.1.1 Definition:

```
LTH:X   == if (X = NIL) then 0
                else if (X = <Y,W>) then 1 +LTH:<W>
S(i):X  == if (X = <Y,W> and i = 1) then Y
                else if (X = <Y,W> and i > 1)
                        then S(i-1):<W>
TL:X    == if (X = <Y,W>) then <W>
PRFX:X  == if (X = <Y,<W>>) then <Y,W>
SUFX:X  == if (X = <<W>,Z>) then <W,Z>
ID:X    == X
ATOM:X  == if (X is an atom) then T
```

```
                else if (X is in ITEM) then F
ISNIL:X == if (X = NIL) then T
                else if (X is in ITEM) then F
```

In addition to these functions, we will make free use of functions implementing the usual arithmetic, relational and logical operators. Whenever it is convenient to do so, we will write these functions in infix notation. Thus +:<x,y> will be written as x + y. (Of course, this requires us to assume the usual precedence of operations and, on occasion, to add parentheses to arithmetic expressions.

The pertinent properties of these functions are summarized in the proposition below. The proofs of these properties are straightforward applications of the definitions of the functions and are therefore omitted. (PHI represents the function which is everywhere undefined.)

2.1.2 Proposition:

```
i)    LTH.PRFX = 1 + LTH.S(2)
ii)   LTH.SUFX = 1 + LTH.S(1)
iii)  LTH.TL = if ISNIL then PHI
                        else LTH - 1
iv)   TL.PRFX = if LTH = 2 & ~ATOM.S(2)
                    then S(2)
```

```
                    else PHI
v)     TL.SUFX = if LTH = 2 & ~ATOM.S(1)
                    then
                      if ISNIL.s(1)
                         then [S(2)]
                         else SUFX.[TL.S(1).S(2)]
                    else PHI
vi)    S(i).PRFX = if LTH=2 & ~ATOM.S(2)
                      then if i=1 then S(1)
                                  else S(i-1).S(2)
                      else PHI
vii)   S(i).SUFX = if LTH=2 & ~ATOM.S(1)
                      then if i=1+LTH.S(1)
                              then S(2)
                              else S(i).S(1)
                      else PHI
viii) S(i).TL = S(i+1)
```

These primitive functions will be combined using the functional forms provided by the language. The list below contains only those functional forms which will be used in this paper. In this list, f and g represent functions which have been extended from ITEM to ITEM+, p represents the extension of a function which maps ITEM into {T,F}, and e represents a string of symbols such that [e] is a valid

functional form. X, Y, Z, and W are as above. It is once again assumed that the function obtained as the result of applying any of these functional forms will be extended to ITEM+ in the usual way.

2.1.3 Definition:

```
    Composition:   (f.g):X   == f:(g:X)
    Construction:  [f]:X     == <f:X>
                   [f,e]:X   == PRFX:<f:X,[e]:X>
```

(Observe that, from our definition of PRFX, if we apply a function of the form [f1,...,fn] to X and fi:X = PHI for some i, then the function PRFX will eventually return PHI as its value and [f1,...,fn]:X = PHI. In the same way, if f:X = PHI, then we must agree that [f]:X = PHI since <PHI> is indeed undefined.)

```
    Condition:     (if p then f else g end):X
                           == if p:X = T then f:X
                                 else if p:X = F then g:X
    Insert:        /f:X     == if X = <Y>
                                 then <Y>
                                 else if (X = <Y,W>
                                        then f:<Y,/f:<W>>
    While:         (while p do f end):X
                           == if p:X = T
```

then (while p do f end):(f:X)

else if p:X = F then X

In addition to these primitive functions and functional forms, we need a mechanism for defining functions and assigning names to them. As part of the definition, we would like to be able to specify the domain and range of the function being defined. For this purpose, we shall adopt the notation:

name == function: < range , domain >, definition

In order to make our definitions as readable as possible, we will occasionally want to supply local names for the selector functions. The scope of these names will be limited to the function definition and, since function definitions will not be nested, no ambiguities will result. We will specify these local names by listing them at the end of the function definition. The meaning of each of the local names will be determined by its position in the list. Thus the first name in the list will be used for S(1), the second for S(2) and so on. The original names of these selectors, for example S(1) and S(2), will still be used in contexts in which the newly assigned names would be meaningless.

2.1.4 Example

The following simple function definitions illustrate the notational conventions which we have adopted.

```
FACTORIAL == function: <int,int>,
             if id = 0
                then 1
                else if id > 0
                     then id*FACTORIAL.(id - 1)
                     end
             end.

BINCOEF   == function: <int, <int,int>>,
             FACTORIAL.R /
               (FACTORIAL.N * FACTORIAL.(N - R) ).
             (N,R)
```

One last notation will be used in the following section on fundamental theorems and in the verifications in the following chapter. It is the notation

```
f = p1 -> g1; p2 -> g2; ... ; pn -> gn; ...
```

where each pi is a predicate and each gi is a function.

When applied to an element, x, if {i ! pi:x = T} is nonempty, then f:x = gk:x where k is the smallest entry in this set. Otherwise, f:x = PHI.

2.2 Fundamental Theorems

As we have observed, functional programming systems demonstrate a close relationship with theoretical results on properties of programs. In this section, we elaborate upon that relationship and present those results which will be used in this paper. In particular, it is worth noting that these results may be presented in the notation and terminology of the system we are using and that this is characteristic of any functional programming system, not simply a peculiarity of the one being used here.

We begin by recalling the definition of monotonicity of functions. A function f is monotonic with respect to some order relation, say *<*, provided that for any x, y, x *<* y implies that f:x *<* f:y. All of the functions we are considering are monotonic with respect to the relation of set inclusion.

Following [7], given two such functions, f and g, we say that f is less defined than or equal to g, and write f < g, provided that for all x, if f:x <> PHI, then f:x = g:x. This implies that the restricted domain of f must be a subset of

the restricted domain of g and that, on the smaller of these two sets, the two functions must be identical. In fact, the relation 'less defined than or equal to' is equivalent to the relation of set inclusion when f and g are viewed as sets of ordered pairs in the usual way. Thus, the following proposition is obvious.

2.2.1 Proposition: If f0 < f1 < f2 < ... < fn < ... is a chain of functions, then f = Ufi is a function such that fi < f for all i. Furthermore, given any function g such that fi < g for all i, we have f < g.

The function defined by the above is the 'smallest' function such that fi < f for all i. Hence it must be unique. It is called the limit of the chain {fi} and is denoted LIMi {fi}.

We will also need several additional definitons. Given a functional T, a function f is called a fixed point of T provided that f = T(f). f is a least fixed point of T if it is a minimal among the set of all fixed points of T. A functional form T is said to be monotonic provided that whenever f < g, T(f) < T(g). T is continuous provided that it is monotonic and, in addition, that given any chain {fi}, T(LIMi {fi}) = LIMi {T(fi)}. Note that continuity of a functional form implies that the functional form is

monotonic. The converse is not true. (See [7,494] for an example of such a functional.)

In order to apply the fixed point theory of [7] it is necessary to demonstrate that all functional forms are continuous. However, since all of the functional forms which are being used are defined by composition of monotonic functions and function variables, this conclusion follows from [7,193]. In addition, a 'fixed point' computation rule must be used. Several such rules are described in [7,496]. We shall not concern ourselves with the particular computation rule being used except to assume that it is a fixed point rule which allows application of the following theorems. In the theorem below, if T is a functional, then T0(PHI) = PHI, and for i >= 0, Ti+1(PHI) represents the function obtained by applying T to Ti(PHI). This is called the First Recursion Theorem in [7] and is due to Kleene.

2.2.2 Theorem: Any continuous functional T has a least fixed point which is the limit of the chain

T0(PHI) < T1(PHI) < ... < Tn(PHI) < ...

The proof of this theorem follows immediately from the observations that PHI < T(PHI) for any functional T; that, whenever Ti-1(PHI) < Ti(PHI), the monotonicity of T guarantees that Ti(PHI) < Ti+1(PHI); and that the

continuity of T assures that T(LIMi(Ti(PHI))) = LIMi(T(Ti(PHI))) = LIMi(Ti(PHI)).

It is, in general, quite difficult to obtain this least fixed point in a useful form. There are, however, several important cases where this can be done. In particular, suppose that E is a functional such that for some sequence of functions (fi) we have (i) f0 = PHI, (ii) fi+1 = p0 -> q0; ... ; pi -> qi; PHI for i = 0,1,... and that E(fi) = fi+1 for all i. Then E is said to be expansive with approximating functions fi. In this case, the least fixed point of E is the function

f = p0 -> q0; ... ; pn -> qn; ... = LIMi(fi).

[1,630]

The utility of this result is illustrated by the following theorem.

2.2.3 Theorem (Recursion Theorem): Suppose f is defined by

f == if p then g else h.[i,f.j] end

for arbitrary functions p, g, h, i, and j. Then

f = p -> g; p.j -> Q(g); ... ; p.j^n -> Q^n(g) ...

where q^1(g) = h.[i,g.j] and Q^n(g) = h.[i,Q^(n-1)(g).j]. Furthermore, Q^n(g) = /h.[i,i.j,...,i.j^(n-1),g.j^n].

[1,627-628]

The following theorem is obtained as a corollary.

2.2.4 Theorem (Iteration Theorem): Suppose f is the least fixed point of f == if p then g else h.f.k end. Then

f = p -> g; p.k -> h.g.k; ... ; p.k^n -> h^n.g.k^n ...

[1,628]

The following is an easy corollary to the Iteration Theorem.

2.2.5 Corollary: If f == while p do k end; then

f = ~p -> id; ~p.k -> k; ... ; ~p.k^n -> k^n; ...

The proof of this corollary follows immediately from the observation that f may be defined as the least fixed point of the functional E given by E(h) = ~p -> id; h.k. In order to prove termination for such a function applied to an item X, it is sufficient to show that (m: ~p.k^m:X) is nonempty.

This result is extremely useful and is therefore stated as a corollary for future reference.

2.2.6 Corollary: If f == while p do k end; then f:X terminates if and only if {m | ~p.k^m:X} is not empty.

3. Program Verification

3.1 Introduction

Since functional programming systems are based on successive application of functions rather than successive state transformations, the cumbersome formalisms necessary for identifying various states, as seen in verification methods for Algol-type languages, are eliminated. Since functional languages do not depend on changing values in a store, the proofs have a much more natural appearance. In addition, we have adopted a number of facilities such as infix operators and local naming of selectors which make these proofs much easier to write and to read.

As an example of the versatility of functional programming systems in the area of program verification, we shall define a parameterized type, SEQ, and functions SORT and SEARCH which can be applied to SEQ:A for any ordered type A. We then demonstrate the correctness of each of these functions.

3.2 The Parameterized Type SEQ

3.2.1 Definition: An ordered type consists of a carrier set, otype, together with a function LE whose domain is the

set <BOOL,<otype,otype>> and which satisfies the following:

i. LE:<a,a> = T for all a in otype,

ii. If LE:<a,b> and LE:<b,a> then a = b for any a,b in otype.

iii. If LE:<a,b> = T and LE:<b,c> = T then LE:<a,c> = T for any a,b,c in otype.

Given the function LT (less than), one can easily define the functions LE, GE, and GT in the obvious way. In the following, we shall use any of the four, as convenience dictates, denoting them by <, <=, >=, and > as usual and writing them as infix operators.

The parameterized type SEQ is defined by

SEQ == function:<type,type>, NIL:Prfx.[id,SEQ].

Intuitively, SEQ accepts any type and returns the type consisting of all sequences of elements of the given type. Thus, for example,

SEQ:INT = NIL ; <INT> ; <INT,INT> ...

This notation allows us to describe functions on various domains involving this parameterized type. For example, the

domain consisting of all possible pairs of sequences of integers may be represented by [SEQ,SEQ]:int.

3.3 SORT

The function SORT will be defined as a selection sort which may be applied to any instance of SEQ applied to some ordered type. We begin by defining a function, MIN, which accepts such a sequence and returns a new sequence which is a permutation of the original sequence having the smallest element first. We shall begin with an iterative definition of this function, but we will also consider a recursively defined version later.

```
MIN == function: <SEQ:otype,SEQ:otype>,
       NEW_SEQ.
         while ~ISNIL.OLD_SEQ do
           [TL.OLD_SEQ,
              if S(1).OLD_SEQ < S(1).NEW_SEQ
                then PRFX.[S(1).OLD_SEQ,NEW_SEQ]
                else SUFX.[S(1).OLD_SEQ,NEW_SEQ]]
         end.
         [TL,[S(1)]]  (OLD_SEQ,NEW_SEQ)
```

Inspection of this definition shows that MIN works by

considering the elements of the original sequence one at a time from left to right. Each of these elements is placed either at the front or the back of the new sequence being formed, as its relative size dictates.

Note that we assume that application of the function MIN to anything other than a sequence of elements from some ordered type will return PHI as its value. Further observe that if ISNIL:A then MIN:A will return PHI.

Assuming, for the moment, that MIN works correctly, we define the function SORT.

```
SORT == function, [SEQ,SEQ]:otype,
        ORDERED_SEQ.
        while ~ISNIL.REST do
          [TL.MIN.REST,
             SUFX.[S(1).MIN.REST,ORDERED_SEQ]
          ]
        end.
        [id,NIL]  (REST,ORDERED_SEQ)
```

We now proceed to state and prove a theorem to establish the correctness of the function SORT. We begin, however, by considering what we need to prove to show that SORT is 'correct'. Clearly, we need to demonstrate that both MIN and SORT terminate. Further, we need to verify that SORT

returns an element of type SEQ:otype which is ordered. This is not, however sufficient, for it would be satisfied if, for example, we had specified that SORT return the empty sequence. Similarly, it is not sufficient to show that every element of A is in SORT:A since the possibility would still exist that SORT might introduce or eliminate duplicate elements. It is apparent, then, that we need to show that SORT:A is an ordered sequence which is, in fact, a permutation of the elements of A. This leads to the following theorems.

We begin with a theorem showing that MIN performs correctly. In addition to proving that it terminates and that the first entry of the sequence returned does not exceed the other entries, we must prove that the returned sequence is simply a permutation of the elements in the original sequence in order to deduce the corresponding result for the function SORT.

3.3.1 Theorem: If A is in SEQ:otype and ~ISNIL:A then, letting n = LTH:A, the following results hold.

i. MIN:A terminates.

ii. For i = 2,...,n, S(1).MIN:A <= S(i).MIN:A.

iii. There is a permutation p of {1,...,n} such that, if A = <X1,...,Xn> then MIN:A = <Xp(1),...,Xp(n)>.

proof: Note that the function MIN may be written as

NEW_SEQ.while ~ISNIL.OLD_SEQ do h end.[TL,[S(1)]]

where

```
h == function:[[SEQ,SEQ],[SEQ,SEQ]]:otype,
     [TL.OLD_SEQ,
        if S(1).OLD_SEQ < S(1).NEW_SEQ
          then PRFX.[S(1).OLD_SEQ,NEW_SEQ]
          else SUFX.[S(1).OLD_SEQ,NEW_SEQ]
     ]   (OLD_SEQ,NEW_SEQ)
```

For j >= 0, let Yj = h^j.[TL,[S(1)]]:A. Then, by 2.2.6,

```
MIN:A = NEW_SEQ.( ISNIL.OLD_SEQ:Y0 -> Y0; ...;
                  ISNIL.OLD_SEQ:Yj -> Yj;...)
```

But, observe that LTH.OLD_SEQ:Y0 = LTH.TL:A = n-1 and

```
LTH.OLD_SEQ:Yj+1 = LTH.TL.OLD_SEQ:Yj
                 = (LTH.OLD_SEQ:Yj)-1.
```

Thus, one can easily show inductively that LTH.OLD_SEQ:Yj = n-(j+1) = n-j-1. Similarly, LTH.NEW_SEQ:Yj = j+1. Therefore, LTH.OLD_SEQ:Yn-1 = n-n = 0 and ISNIL.OLD_SEQ:Yn-1 = true. Thus, it is apparent that MIN terminates when applied to any non-NIL element of type SEQ:otype and returns the value

```
NEW_SEQ:Yn-1 = NEW_SEQ.h^(n-1).[TL,[S(1)]]:A.
```

We may complete the proof of (ii) by showing inductively that S(1).NEW_SEQ:Yj <= S(i).NEW_SEQ:Yj for i = 2,...,j+1 and j = 0,...,n-1. For j = 0, this result is vacuously true. Assume then that it is true for some j, 0 <= j < n-1. Then

```
Yj+1 = h:Yj =
[TL.OLD_SEQ:Yj,
   if S(1).OLD_SEQ:Yj < S(1).NEW_SEQ:Yj
     then PRFX:[S(1).OLD_SEQ,NEW_SEQ]:Yj
     else SUFX:[S(1).OLD_SEQ,NEW_SEQ]:Yj
]  (OLD_SEQ,NEW_SEQ)
```

Thus, we consider the following two cases.

Case I: S(1).OLD_SEQ:Yj < S(1).NEW_SEQ:Yj

In this case,

```
S(1).NEW_SEQ:Yj+1 = S(1).PRFX.[S(1).OLD_SEQ,NEW_SEQ]:Yj
                  = S(1).OLD_SEQ:Yj
```

by 2.1.2 (vi).

But

S(1).OLD_SEQ:Yj < S(1).NEW_SEQ:Yj <= S(i).NEW_SEQ:Yj

for i = 1,...,j+1 by the induction hypothesis.

For i = 2,...,j+1,

```
S(i).NEW_SEQ:Yj = S(i).PRFX.[S(1).OLD_SEQ,NEW_SEQ]:Yj
                = S(i-1).NEW_SEQ:Yj
```

by 2.1.2 (vi). But, if s <= i <= j+2 then 1 <= i-1 <= j+1.

Thus we may conclude that

S(1).NEW_SEQ:Yj+1 <= S(i).NEW_SEQ:Yj+1

for i = 2,...,(j+1)+1.

This concludes the inductive step for Case I.

Case II: S(1).OLD_SEQ:Yj >= S(1).NEW_SEQ:Yj

Now,

NEW_SEQ:Yj+1 = SUFX.[S(1).OLD_SEQ,NEW_SEQ]:Yj.

Thus

S(i).NEW_SEQ:Yj+1 = S(i).NEW_SEQ:Yj

for i = 1,...,j+1 and

S(j+2).NEW_SEQ:Yj+1 = S(1).OLD_SEQ:Yj

by 2.1.2 (vii). The desired result follows immediately from the induction hypothesis.

This completes the proof of part ii.

In order to establish iii, it will suffice to show that for each j = 0,1,...,n-1, there is a permutation pj of (1,...,j+1) such that NEW_SEQ:Yj = <Xpj(1),...,Xpj(j+1)>. Since NEW_SEQ:Y0 = <X1>, this is true for j = 0. Then, observing that OLD_SEQ:Yj = <Xj+2,...,Xn> and assuming the result to be true for some j >= 0, we construct pj+1 as follows:

For Case I:

Yj+1 = [TL.OLD_SEQ:Yj,

PRFX:[S(1).OLD_SEQ:Yj,NEW_SEQ:Yj]]

Thus NEW_SEQ:Yj+1 = <Xj+2,Xpj(1),...,Xpj(j+1)> and pj+1 can be defined by

pj+1(i) == if i = 1
then j+2
else pj(i-1)

for i = 1,...,j+2.

For Case II:

Yj+1 = [TL.OLD_SEQ:Yj,
SUFX:[S(1).OLD_SEQ:Yj,NEW_SEQ:Yj]]

NEW_SEQ:Yj+1 = <Xpj(1),...,Xpj(j+1),Wj+2>

and we define pj+1 by

pj+1(i) == if 1 <= i <= j+1
then pj(i)
else j+2

for i = 1,...,j+2.

Then, in either case,

NEW_SEQ:Yj+1 = <Xpj+1(1),...,Xpj+1(j+2)>

for some permutation pj+1 of {1,...,j+2}.

MIN:A = NEW_SEQ:Yn-1 = <Xpn-1(1),...,Xpn-1(n)>. Thus, pn-1 is the required permutation.

This concludes the proof of Theorem 3.3.1.

We observe, as a corollary to this theorem, that LTH.MIN:A = LTH:A.

We now state and prove a theorem assuring that the function SORT performs correctly. It is important to observe during the proof of this theorem that it is never necessary to know how the function MIN works or even to know exactly what value will be returned. The only information about MIN that is required is that which is contained in Theorem 3.3.1.

3.3.2 Theorem: Let A be in SEQ:otype. Then A = <X1,...Xn> for some n >= 0 and X1,...,Xn in otype.

i. SORT:A terminates.

ii. For i = 1,...,n-1, S(i).SORT:A <= S(i+1).SORT:A.

iii. There exists a permutation p of (1,...,n) such that SORT:A = <Xp(1),...,Xp(n)>.

proof: If we let

```
h == function:[SEQ,SEQ]:otype,
       [TL.MIN.REST,
         SUFX.[S(1).MIN.REST,ORDERED_SEQ]
       ]    (REST,ORDERED_SEQ)
```

then

```
SORT = ORDERED_SEQ.
         while ~ISNIL.REST do h end.
         [id,NIL].
```

LET Y_j = h^j.[id,NIL]:A for j >= 0. Then, by 2.2.6,

SORT:A = ORDERED_SEQ.

(ISNIL.REST:Y0 -> Y0;...;

ISNIL.REST:Yj -> Yj;...).

Observing that LTH.REST:Y0 = n and

LTH.REST:Yj+1 = LTH.REST.h:Yj

= LTH.TL.MIN.REST:Yj

= (LTH.MIN.REST:Yj)-1,

We conclude inductively that LTH.REST:Yj = n-j for j = 0,...,n. Thus SORT:A terminates returning the value REST:Yn. (Note that, in the same way, one can show that LTH.ORDERED_SEQ:Yj = j.)

In order to establish (ii), we show that

S(i).ORDERED_SEQ:Yj <= S(i+1).ORDERED_SEQ:Yj

for i = 1,...,j-1 and for j = 1,...,n by induction on j.

For j = 1, the inequality is vacuously true. Assume true for some j, 1 <= j < n and consider

ORDERED_SEQ:Yj+1 = ORDERED_SEQ.h:Yj

= SUFX.[S(1).MIN.REST,ORDERED_SEQ]:Yj

Therefore, for i = 1,...,j,

S(i).ORDERED_SEQ:Yj+1

= S(i).SUFX.[S(1).MIN.REST,ORDERED_SEQ]:Yj

= S(i).ORDERED_SEQ:Yj

by 2.1.2 (vii). Hence, the induction hypothesis guarantees

that

S(i).ORDERED_SEQ:Yj <= S(i+1).ORDERED_SEQ:Yj+1

for i = 1,...,j-1 and we need only prove that

S(j).ORDERED_SEQ:Yj+1 <= S(j+1).ORDERED_SEQ:Yj.

As above,

S(j).ORDERED_SEQ:Yj+1 = S(j).ORDERED_SEQ:Yj

= S(j).ORDERED_SEQ.h:Yj-1

= S(j).SUFX.[S(1).MIN.REST,

ORDERED_SEQ]:Yj-1

= S(1).MIN.REST:Yj-1

by 2.1.2 (vii), since LTH.ORDERED_SEQ:Yj-1 = j-1.

While

S(j+1).ORDERED_SEQ:Yj+1 = S(j+1).SUFX.

[S(1).MIN.REST,

ORDERED_SEQ]:Yj

= S(1).MIN.REST:Yj

as above.

But,

S(1).MIN.REST:Yj = S(1).MIN.TL.MIN.REST:Yj-1.

and, by 3.3.1, the elements of MIN.TL.MIN.REST:Yj-1 are

permutations of the elements of TL.MIN.REST:Yj-1.

Therefore,

S(1).MIN.REST:Yj = S(k).TL.MIN.REST:Yj-1

for some k in (1,...,j-2).

Thus, by 2.1.2 (viii),

S(1).MIN.REST:Yj = S(k+1).MIN.REST:Yj-1,

with 2 <= k+1 <= j-1. But then, by 2.3.1,

S(1).MIN.REST:Yj-1 <= S(k+1).MIN.REST:Yj-1.

Thus we may conclude that

S(j).ORDERED_SEQ:Yj+1 <= S(j+1).ORDERED_SEQ:Yj+1

which completes the induction.

Part iii of the theorem may be established by proving inductively that for j = 0,1,...,n, the sequence obtained by concatenating REST:Yj with ORDERED_SEQ:Yj is a permutation of the elements of A. This is a straightforward application of the corresponding part of Theorem 3.3.1 and is therefore omitted.

This concludes the proof of Theorem 3.3.2.

We should, perhaps, pause at this point to emphasize one of the most important aspects of program verification in functional programming systems. We are guaranteed that separate functions can never interfere with one another. Thus we can replace any component of a large system with a new version, provided that we can demonstrate that the new version produces equivalent results. It is never necessary to concern ourselves with the way in which these results are obtained.

As we observed earlier, we could have defined the function MIN recursively. The verification of the function

SORT required no knowledge of the way in which the function MIN operated. Thus we may make such a change without fear of affecting SORT provided only that we verify that the new version of MIN also satisfies Theorem 3.3.1. Consider then the following recursive version of MIN.

```
MIN2 == function:[SEQ,SEQ]:otype,
        if ISNIL.TL
             then ID
             else if S(1) < S(1).S(2)
                       then PRFX
                       else SUFX
                  end.[S(1),MIN.TL]
         end.
```

The proof that this function also satisfies Theorem 3.3.1 is a simple exercise. Of particular interest to us, however, is the fact that although this function may replace MIN in our definition of SORT, it is not the same function. For example, MIN:<3,1,4,2> returns the sequence <1,3,4,2> whereas MIN2:<3,1,4,2> returns <1,2,4,3>. Our approach then has made quite explicit the properties which a function MIN must satisfy in order that SORT work properly. Beyond making certain that our implementation of MIN satisfies these properties, we are quite free, as the examples above

demonstrate, to choose any implementation that we find to be convenient.

3.4 Binary Search

We now consider a function to perform a binary search on an element of type SEQ:otype which has been ordered by SORT. Obviously, in order to justify the use of a binary search, we need some random access mechanism. Such a mechanism may be provided in a functional programming system, but for our purposes, we may simulate it with the following function.

```
SEL == function:<type,<SEQ:type,INT>>,
       S(1).SEQUENCE.
       while INDEX > 1 do
         [TL.SEQUENCE,INDEX-1]
       end (SEQUENCE,INDEX)
```

It is apparent that if A = <X1,...,Xn> then for i = 1,...,n, SEL:<A,i> = S(i):A = Xi.

We now give the binary search function. As in the case of the function SORT, we find it convenient to define a preliminary function.

```
MID == function:<INT,<SEQ:otype,INT,INT,otype>>,
```

```
          (B+C)/2
    SEARCH == function:<INT,<SEQ:otype,INT,INT,otype>>,
              B.
              while B < C do
                if KEY > SEL.[A,MID]
                  then [A,MID+1,C,KEY]
                  else [A,B,MID,KEY]
              end (A,B,C,KEY).
```

(Note that the division in MID is assumed to be integer division.)

3.4.1 Theorem: If A0 = <A1,...,Xn> is an ordered sequence and X, B0, C0, and J are integers such that

    1 <= B0 <= J <= C0 <= n

and X = S(J):A0, then SEARCH:<A0,B0,C0,X> terminates returning the value J.

proof: Let g be the function given by

```
    g == function:<<SEQ:otype,INT,INT,otype>,
                   <SEQ:otype,INT,INT,otype>>,
         if KEY > SEL.[A,MID]
            then [A,MID+1,C,KEY]
            else [A,B,MID,KEY]. (A,B,C,KEY)
```

and let p be the predicate B < C. Then

```
    SEARCH = B.while p do g end
```

and, by 2.2.6,

SEARCH = B.(~p -> ID; ~p.g -> g;...; ~p.g^n -> g^n;...).

Let Yj = g^j:<A0,B0,C0,X>. One can easily verify that (B-C):Yj+1 < (B-C):Yj. This is sufficient to assure termination. Thus SEARCH:<A0,B0,C0,X> = B:Ym for some m >= 0.

We now prove that (B <= J <= C):Yj for 0 <= j <= m by induction on j. For j = 0,

(B <= J <= C):Y0 = (B0 <= J <= C0)

which is true. Now, assume the result is true for some j, 0 <= j <= m. Since (B <= J <= C):Yj and j<m, (B < C):Yj and, since MID = (B+C)/2, we may conclude (B <= MID <= C):Yj.

Now consider Yj+1 = g:Yj. If (KEY > SEL.[A,MID]):Yj, then, since A0 is ordered, (MID < J):Yj or (MID +1 <= J):Yj. Therefore,

B:Yj+1 = (MID +1):Yj <= J <= C:Yj = C:Yj+1

and

(B <= J <= C):Yj

as required.

On the other hand, if (KEY <= SEL.[A,MID]):Yj then (J <= MID):Yj and

B:Yj+1 = B:Yj <= J <= MID:Yj = C:Yj+1.

Thus, we again have (B <= J <= C):Yj+1 completing the

induction.

We may therefore conclude that (B <= J <= C):Ym. But, (B < C):Ym is false because of the choice of m. Hence (B = C):Ym and J = B:Ym = SEARCH:<A0,B0,C0,X> completing the proof of the theorem.

4 Conclusions

4.1 Observations

As we noted in Chapter 1, the proof techniques demonstrated in Chapter 3 relied upon the theory of least fixed points of continuous functionals to prove termination. The proofs of partial correctness used results obtained from the fixed point theory together with a variation of structural induction. This is not to imply that other verification methods cannot be applied to functional programming systems. Indeed, at the conclusion of this chapter, we shall consider the application of one of the classical techniques, Dijkstra's weakest precondition methods for program synthesis, and see that, not only can these be applied with only minor modifications, but that their application appears to be much more direct than is the case with Algol-type languages.

In addition to eliminating the problems of representing values in a changing store as part of a verification, functional programming systems provide two other very important features. They completely eliminate the possibility of any side effects and remove the problem of representing the semantic effect of procedure calls involving other than value parameters. These are precisely the areas which present

the most difficulty for techniques such as Hoare's axiomatic approach and Dijkstra's synthetic techniques. Furthermore, it is quite natural to separate the the iterative or recursive parts of a program into relatively simple functions by the simple expedient calling another function whenever they are needed. This allows a clear separation in the proof between the necessity of proving termination and the proof of partial correctness of the function.

The advantages of these changes may clearly be seen in the proofs of the theorems in Chapter 3. Thus, for example, when proving Theorem 3.3.1, we rewrote MIN in the form

```
NEW_SEQ.while ~ISNIL.OLD_SEQ do h end.[TL,[S(1)]].
```

Then the question of termination could be settled by considering the simpler function

```
while ~ISNIL.OLD_SEQ do h end.
```

Once termination has been established, the original function may be examined using an axiomatic approach based on the properties of the primitive functions and functionals of the programming system and previously established properties of user defined component functions.

It should be noted that there is more involved here than

the usual benefits which accrue as a result of designing modular programs in an Algol-type language. In a functional programming system, we can enjoy the benefits of modularity without having the increased complexity of modeling procedure calls and parameter passing in the verification process.

Of course, when applied on a small scale as in these sample verifications, such effects are merely minor conveniences. On a larger scale, however, these effects may well determine the feasibility and practicality of such a program verification system.

4.2 Weakest Preconditions

Finally, we can make use of synthetic methods such as Dijkstra's weakest precondition techniques to aid in the design of programs in a functional programming system. Such techniques are designed to accept the specifications for a program and produce both the program and its verification. This may become one of the most important aspects of functional programming systems since it may lead to automatic or semi-automatic programming systems. Thus we conclude by considering the application of Dijkstra's methods to a functional programming system.

Suppose that we need to construct a function f satisfying a loop invariant g such that the loop construct

```
    while p do f end
```

terminates over the set D = {i : g:i}. Given a collection h1, h2, ..., hn of possible alternatives for f and a non-negative integer valued function t, let

```
    pi = (t.hi < t) & (g.hi)
```

for i = 1 to n. If the implication p -> (p1 : p2 : ... : pn) is true over the set D' = {x in D : p:x}, then

```
    f == if p1 then h1
            else if p2 then h2
                    .
                    .
                        else hn
```

satisfies the requirements. (The proof is a straightforward induction on the values of t.)

As an example of the use of this approach, we consider a simplified analog of the example found in Chapter 7 of [4]. We attempt to develop and verify a program, GCD:<x,y>, to compute the greatest common divisor of any pair of positive integers using as our only information about the greatest common divisor the following equations:

i. GCD:<x,y> = GCD:<y,x>.

ii. GCD:<x,y> = GCD:<x+y,y> = GCD:<x-y,y>, etc.

iii. GCD:<x,y> = x if x = y.

Following Dijkstra, we observe that only the last of these equations gives the GCD in a finalized form. Thus our program must apparently do something to the values x and y to reduce the initial problem to the form of equation iii. If we assume that we can write this program as a simple iteration, then whatever we decide to do to x and y, our program would be of the form

GCD == X.while p do f end (X,Y)

where p is the predicate ~(X = Y) and f is some, as yet unknown, function such that f:<x,y> has the same greatest common divisor as <x,y>. Our loop invariant is that for any initial pair <x0,y0> of integers each application of f must return a pair of positive integers with the same greatest common divisor as <x0,y0>. We can ensure the latter by using only the transformations from equations i, ii, and iii as the alternatives h1, ... , hn. Thus, we can use the formula

pi = (t.hi < t) & ((X > 0) & (Y > 0)).hi

with the alternatives h1 = [Y,X], h2 = [X+Y,Y], h3 = [X-Y,Y], h4 = [X,Y+X], h5 = [X,Y-X].

Choosing t = X + Y, and calculating each of the functions, pi, over the domain determined by the loop

invariant, we find that p1, p2, and p4 are always false.

```
p3 = ((X+Y).[X-Y,Y] < (X+Y))
                & ((X > 0) & (Y > 0)).[X-Y,Y]
   = ( (X) < (X + Y)) & ((X-Y) > 0) & (Y > 0)
   = ( 0 < Y) & (X > Y) & (Y > 0)
   = (X > Y) & T  = (X > Y)
```

Similarly, p5 = (Y > X). Thus, p -> (p3 ; p5) for all values in this domain and f = if p3 then h3 else h5 satisfies our requirements.

We have thus derived and verified the function definition

```
GCD == X.
       while ~(X = Y) do
         if (X > Y) then [X-Y,Y]
                    else [X,Y-X]
       end (X,Y)
```

4.3 Summary

Functional programming systems provide a number of advantages for program verification. As the sample verifications presented here show, these include a close relationship with the theoretical foundations of program verification and the complete absence of side effects, alias problems and the problems of representing the semantics of

parameter passing. This results in verifications which demonstrate a much closer relationship to the program being verified than is possible in non-functional programming systems.

References

1. Backus, John W., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Comm. ACM 21,8 (Aug. 1978), 613-641.

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

3. Dershowitz, Nachun and Manna, Zohar. Proving termination with multiset orderings. Comm. ACM 22,8 (Aug. 1979), 465-476.

4. Dijkstra, Edsger W. A Discipline of Programming, Prentice-Hall, Inc. Englewood Cliffs, N. J., 1976.

5. Floyd, R. W. Assigning meanings to programs. Proc. Symposium in Applied Mathematics, Vol. 19, J. T. Schwartz (Ed.), Amer. Math. Soc., pp 19-32, 1967.

6. Hoare, C. A. R. An axiomatic basis for computer programming. Comm. ACM 12,10 (Oct. 1969), 576-580, 583.

7. Manna, Zohar, Ness, Stephen, and Vuillemin, Jean. Inductive methods of proving properties of programs. Comm. ACM 16,8 (Aug. 1973), 491-502.

8. Manna, Zohar and Vuillemin, Jean. Fixpoint approach to the theory of computation. Comm. ACM 15,7 (Jul. 1972), 528-536.

9. Martin, Johannes J. FAD, a functional language that supports abstract data types. Proceedings of the 1980 Annual Conference of the ACM, (Oct. 1980), 247-262.

10. Morris, James H. Jr. and Wegbreit, Ben. Subgoal induction. Comm. ACM 20,4 (Apr. 1977), 209-222.

The vita has been removed from the scanned document

PROGRAM VERIFICATION IN FUNCTIONAL PROGRAMMING SYSTEMS

by

James Leland Silver, Jr.

(ABSTRACT)

Functional programming systems provide a number of features which facilitate program verification. Such verifications may be observed to rest directly upon the theoretical foundations of computing and simultaneously to exhibit a close relation to the programs being verified.

In order to demonstrate these aspects of functional systems, two functions, MIN and SORT, are defined on a parameterized type consisting of sequences of elements from some ordered type. Theorems showing that MIN and SORT terminate and return the correct values are stated and proved. Similar results are derived for a function to perform a binary search on an ordered sequence.

Finally, conditions similar to Dijkstra's weakest preconditions are given which allow the simultaneous synthesis and verification of certain programs from program specifications. A function to find the greatest common divisor of two integers is derived and verified.