Synthesizing Least Fixed Point Queries into Non-recursive Iterative Programs

Shamim A. Naqvi
Bell Laboratories
Murray Hill, NJ 07974
Lawrence J. Henschen
Northwestern University
Evanston, IL 60201

Abstract

An algorithm is presented which takes a least fixed point query expressed by a basis and inductive step and transforms it into a non-recursive iterative program. By augmenting a relational system with this algorithm, least fixed point queries can be handled automatically, and there is no need to proceduralizc relational algebra for such queries.

1. Motivation

We present an algorithm for synthesizing a non-recursive, iterative program from a Least Fixed Point query (defined later). There are three reasons why such a synthesis will be of interest to researchers in AI in general, and to researchers in program synthesis in particular:

1. It (efficiently) solves a real world problem of practical utility for database workers.
2. The method was derived using techniques from logic programming systems. In particular the backward-chaining depth-first problem solving metaphor leads to an intuitive understanding of the algorithm. Thus a link between logic programming and program synthesis is illustrated.
3. Finally, it illustrates the idea that if programs can be classified into domains (e.g. sort programs, list processing programs etc.) then may be we can derive algorithms for synthesizing programs within a domain.

2. Background

Consider a relational expression of the form

$$R - f(R) \qquad \text{—(I)}$$

(Throughout this paper we shall assume that the degree of the left and right hand sides of all such equations is the same). A LFP of (1) is a relation R^* such that $R^* - f(R^*)$ and R^* C S for any S satisfying (1). Examples of such LFP operations are calculating the number of flights between two cities during a given time period, finding the lowest-level manager common to a group of employees, and determining whether there is an active circuit connecting two points. None of the above can be couched in relational algebra [AHOI.

In studying the question of how powerful a data query language should be, Aho and Ullman [AHOI postulated two general principles and noted that Least Fixed Point (LFP) queries satisfy those principles but cannot be supported by the traditional relational calculus [CODD]. Since in general a LFP of (1) may not exist [TARS], a relation R can be constructed inductively by specifying the basis and inductive rules:

$$\begin{aligned} R_0 &\subseteq R \qquad \text{---(2)} \\ f(R) &\subseteq R \end{aligned}$$

To support LFP queries, [AHO] recommends the addition of iteration and assignment operators to relational algebra. Thus (2) can be written as

```
$R \leftarrow R_0$
do
    $R' \leftarrow R$                ---(3)
    $R \leftarrow R \cup f(R)$
while ($R' \neq R$)
```

In [AHOI it is further shown that if in (2) there is only one occurrence of R in f(R) then certain optimizations can be performed in the calculation of the LFP through (3). In this paper we present a method of automatically synthesizing a *non-recursive while-loop* program, i.e. a program like (3) except that it will be non-recursive, from a LFP query of the form of (2). Such a transformation would have the following points of interest for database researchers:

1. The non-procedural nature of relational algebra is preserved although it would take a fairly sophisticated user to write a LFP query.
2. No theorem-proving is required to support LFP queries as in [WRIG] though our method is grounded in the resolution principle of Robinson [ROB].

3. The program derived for (2) is non-recursive.
4. With a slight abuse of notation of (2), link queries (defined later) of QBE [ZLOO] can also be supported in the same framework.

3. Intuitive Explanation of the Algorithm

It is simpler to develop and explain the algorithm by assuming that the LFP query will be written in relational calculus. Thus, equation (2) becomes

$$\begin{aligned} &R \leftarrow R_0 && \text{-Basis step} \\ &R \leftarrow f(R) && \text{-Inductive step} \end{aligned}$$

As an example, consider a database containing the relations father(x,y) and parent(u,v) for particular values of x, y, u and v. Then a query asking for (certain kinds of) ancestors of a person V can be stated as follows:

$$\begin{aligned} &\text{ancestor}(w,a) \leftarrow \text{father}(w,a) && \text{---(4)} \\ &\text{ancestor}(x,a) \leftarrow \text{ancestor}(x,y) \wedge \text{parent}(y,a) \end{aligned}$$

A few comments on the notation are in order. First, all variables are assumed to be universally quantified. Second, the syntax, on purpose, is like that of PROLOG [PERE]. Third, any base predicate (i.e., a relation occurring in the database) with at least one argument instantiated to a constant represents a retrieval request. Thus, the expression parent(w,a) can be read as S2-a parent", select from the parent relation those tuples whose second components have the value V. Fourth, we have purposefully chosen a somewhat obscure and incomplete definition of ancestors. This is to highlight a distinction later. Finally, our method of handling LFP queries (as described below) has two advantages over a PROLOG system. It not only works for left-recursive assertions but is also independent of the order of the assertions. In fact, putting the above query to a PROLOG system will cause an infinite computation.

In order to proceed with an intuitive description of our method, we need to specify how a backward-chaining problem-solving system works. Given an assertion of the form

$$C \leftarrow g1 \wedge g2 \wedge g3 \;.....$$

and a problem, C, such a problem-solving system matches the problem to the head, i.e. left-hand side of the assertion. If the match succeeds then the original problem, C, is replaced by the subproblems g1, g2,..... i.e. the body of the assertion, with the matching substitution applied to the subproblems. If the head of more than one assertion matches a problem then the system has to make a choice.

Now, assume we have a backward-chaining problem-solving system and, further, that a problem p(x,y,....) is solvable only if at least one of the variables x,y,... is instantiated to a constant. Finally, it is evident that a LFP query can be replaced by a view. Thus, (4) can be replaced by the query

$$\text{ancestor}(w,a) \qquad \text{---(5)}$$

on the view

$$\begin{aligned} &\text{ancestor}(u,v) \leftarrow \text{father}(u,v) && \text{---(6a)} \\ &\text{ancestor}(x,z) \leftarrow \text{ancestor}(x,y) \wedge \text{parent}(y,z) && \text{---(6b)} \end{aligned}$$

We now describe how a backward-chaining problem-solving system would solve (5) using (6a) and (6b). From this description, our method will become obvious.

To solve (5), we match it with the head of (6a) and (6b). Since (5) matches both the assertions, let us choose (6a) first. The matching substitution is (u— w,v —a}, and, thus, the original problem is replaced by the body of (6a), namely, father(w,a). This subproblem is solvable and, in fact, yields a first set of answers which we can collect in a relation, say ancestor. From (5) and (6b), the original problem is replaced by the body of (6b) with the matching substitution applied. This yields ancestor(w,y), parent(y,a). Proceeding in a depth-first manner, the subproblem ancestor(w,y) is not solvable, but parent(y,a) is and yields a set of values for y. These values for y are substituted back into the failed subproblem ancestor(w,y) and this subproblem is asked again. Once again we have a choice between (6a) and (6b) and the above process repeats.

The above iterative pattern can be captured in the following steps:

1. Retrieve father(w,a) and insert these tuples into the answer relation, ancestor.
2. Stack the value V.
3. Pop a value from the stack and assign it to V.
4. Retrieve parent(y,z) and use these new values of y to retrieve father(w,y). Also stack these values of y. Collect tuples retrieved for father(w,y) in ancestor.
5. Go to step 3.

The corresponding program form is:

```
/* all variables accept a set of values */

    ancestor ∪ eval(father(w,a))
    insert-in-queue(Q,a)
    while (Q ≠ empty) do
        z ← remove-from-queue(Q)      ----(A)
        eval(parent(y,z))
        ancestor ∪ eval(father(w,y))
        insert-in-queue(Q,y)
    od
```

Insert-in-queue(Q,a) inserts the value V into queue Q if 'a' has not been inserted into Q before in the current computation. Finally, removc-from-queue(Q) removes the top element of Q.

Let us now consider a right-recursive definition of the above LFP query.

```
ancestor(w,a)                                  ---(7)
ancestor(u,v) ← father(u,v)                    ---(8)
ancestor(x,z) ← parent(x,y) ∧ ancestor(y,z)    ---(9)
```

Once again, a similar problem solver would solve the above query as follows: (7) and (8) yield the primitive subproblem father(w,a). From (7) and (9), to solve ancestor(w,a) solve parent(w,y) and ancestor (y,a). Clearly parent (w,y) cannot be effectively solved because we do not know either 'w' or 'y' To solve ancestor(y,a) reuse (8). This yields a value for 'y' which makes parent (w,y) solvable. In this case the program form is as follow:

```
ancestor ∪ eval(father(w,a))
insert-in-queue(Q,a)
while (Q ≠ empty) do
        z ← removc-from-queuc(Q)
        eval(father(y,z))                ---(B)
        ancestor ∪ eval(parent(w,y))
        insert-on-queue(Q,y)
od
```

Program forms (A) and (B) are similar. In order to illustrate the difference we make the following definitions. Consider a LFP query written as

```
f1 ∧ f2 ∧ ... → R                                      ---(10a)
a1 ∧ a2 ∧ ...∧ R∧....∧ bn ∧ bn-1∧ ...∧ b1→ R
```

Let us call the expression ---(10b)

a1 ∧ a2∧ ...∧ bn ∧ bn-1∧ ...∧ b1

the inductive residue (IR), and the expression

f1 ∧ f2 ...

as the basis residue (BR). Thus from (5), (6a) and (6b) the basis residue is father(u,v) and the inductive residue is parent(y,z). Now notice that the program forms (A) and (B) differ in the order in which the basis and inductive residue expressions are used. If we call the constants specified in the basis step R_0 —> R as driver constants, we are lead to the general program form (C) shown in Fig. 1.

The only thing left to explain is the order in which BR and IR expressions are to be used. This order can be elucidated as follows. Consider an LFP query stated as (10a) and (10b). Hypothesize solving R in (10a) by using (10b). This generates the subproblems

a1∧a2∧..∧R∧..∧b1

Now, by examination, if the subproblem R in the above expression is solvable (i.e. at least one of its variables is known) then the order is "eval(IR); R U eval(BR)'', otherwise it is "eval(BR); R U eval(IR)".

```
R ∪ eval(BR)
insert-in-queue(Q,driver constants)
while (Q ≠ empty) do
        z ← remove-from-queue(Q)
        eval (BR or IR)                  ---(C)
        R ∪ eval(IR or BR)
        insert-in-queue(Q,new driver constants)
 od
```

Figure 1: General Program Form

4. Algorithm

Input: A LFP query written in the form of a basis step

R ← f1 ∧ f2 ∧ ....

and an inductive step

R ← a1 ∧ a2 ∧ ... ∧ R ∧ bn ∧ ... ∧ b1

Output: A program of the form of (C) above.

Method:

1. Find the Basis Residue (BR) and the Inductive Residue (IR).
2. Decide on the order of the BR and IR expressions as follows: Match the head of the BR expression (11) to the head of the IR expression (12). Apply the matching substitution to the body of (12) and if the subproblem, R, in the body of (12) is solvable then the order is {eval(IR); R U eval(BR)) else the order is (eval(BR); R U eval(IR)).
3. Substitute the BR and IR expressions (in the right order) into the program skeleton (C).

A proof that the program form above produces correct answers is given in [HEN]. Briefly, correct answers are those values logically implied by the definitions. Our process is essentially supported resolution which is sound and complete, so that all and only answers are generated. Clearly, if the relation R is not cyclic, the stack must become empty at some point. Certain kinds of cyclicity (e.g. reflexivity) can be easily recognized by slight additions to the algorithm. In any case, a more elaborate termination test for the while-loop, based on remembering the derivation of each value added to R, yields termination in all cases. See [HEN] for details.

The transformation algorithm itself has complexity based on identifying cycles in a (clause connectivity) graph and on unification, both of which admit efficient algorithms. The general case allows for programs representing non-tail recursive definitions as well. In these cases, the eval(IR) part of the loop includes an expanding formula corresponding to the non-tail recursive part of the definition.

Although, we give preference to select operations over joins, this is not required. However, most of the time it is obviously more efficient. For example, in the expression

parent(x1,x2) & parent(x2,x3) & parent(x3,x4)
& father(x4,a)

it makes no sense to form the join of parent(xl,x2) and parent(x2,x3) to start with. A feature of the method is that the form of the database requests is known beforehand so that optimizing these requests on the basis of the physical organization of the database as well as on the basis of redundancy in the set of requests could be carried out when the program is synthesized.

5. Conclusion

We have presented a method which, without using theorem-proving, transforms a LFP query expressed as a recursive definition into a non-recursive program. The use of this method permits one to process LFP queries posed in a conventional relational calculus.

6. Acknowledgements

Dan Fishman pointed out some errors and omissions in an earlier draft of this paper. Earlier drafts were also read and commented upon by Kevin Wilkinson, John Linderman and an annonymous referee at Bell Laboratories. We thank all of them.

REFERENCES

[AHO] Aho, A. and Ullman, J.: Universality of Data Retrieval Languages, Conf. on POPL, 1978, pp. 110-120.

[CODD] Codd, E.: Relational Completeness of Database Sublanguages, in Data Base Systems (ed. R. Rustin), Prentice Hall, Englewood Cliffs, NJ.

[HEN] Henschen, L. and Naqvi, S.: On Compiling Queries in Recursive First-Order Databases, to appear in the Journal of the ACM.

[PERE] Pereira, L., Pereira, F. and Warren, D.: Users Guide to DEC 10 PROLOG, Dept. of AI, Univ. of Edinburgh, 1978.

[ROB] Robinson, J.: A Machine-Oriented Logic Based on the Resolution Principle, Journal of the ACM, 12 1, 1965

[TARS] Tarski, A.: A Lattice-Theoretical Fixpoint Theorem and its Applications, Pacific Journal of Mathematics 5:2, 1955.

[WR1G] Wright, D.: PROLOG as a Relationally Complete Database Query Language which can handle Least Fixed Point Operators, Univ. of Kentucky Tech. Report no. 73-80.

[ZLOO] Zloof, M.: Query-By-Example: A Database Management Language, IBM Systems Journal 16, no. 4, 1977.

[ZL002] Zloof, M.: Query-By-Example: Operations on the Transitive Closure, IBM Technical Report no. 5526.

DIAGNOSTIC REASONING IN SOFTWARE FAULT LOCALIZATION

Robert L. Sedlmeyer*
William B. Thompson
Paul E. Johnson

University of Minnesota

ABSTRACT

We present a diagnostic model of software fault localization. A *diagnoatic.* approach to fault localization has proven effective In the domains of medicine and digital hardware. Applying this approach to the software domain requires two extensions: a heuristic abstraction mechanism which infers program function from structure using recognition and transformation tactics; and a search mechanism which integrates both prototypic and causal reasoning about faults.

I. INTRODUCTION

In this paper we present a model of fault localization for program debugging based on a trouble-shooting paradigm [J], Within a diagnostic framework we define the. software fault localization task as follows: A fault exists In a program whenever its output differs from that expected by the user. These descrepancies are called fault *manifestations.* The task is to identify the cause of each manifestation which is specific enough to effect program repair.

Application of the trouble-shooting strategy has proven effective in constructing intelligent systems for hardware [2,3,4] and medical [5,6,7] diagnosis. In contrast to verification-based approaches (cf. [8,9,10,11] this strategy concentrates computational resources on suspect components. Applying the diagnostic approach to the software domain requires two extensions: a heuristic abstraction mechanism that infers function from structure using recognition and transformation tactics; and a search mechanism that integrates both prototypic and causal reasoning to localize faults.

Several program debugging systems evidence trouble-shooting tactics [12,13,14], but none are based on a comprehensive diagnostic theory. Sussman [1] was primarily interested in developing a theory of skill acquisition and examined the role of debugging in that context. While Miller and Goldstein [13] addressed the debugging task directly, the faults which MYCROFT analyzed were tightly constrained by the simplicity of the programs Involved. Shapiro [14] addresses more realistic debugging environments, but formulates a theory of faults rather than of fault localization.

*Current address: Department of Computer Tech., Purdue University at Fort Wayne, Indiana 46805

Our model partitions diagnostic knowledge for the software fault localization task into knowledge uspd to locate *known* and *novel* faults. For diagnosing *known* faults the knowledge base contains a hierarchy of faults which are known to occur in programs from a particular applications domain, and a set of empirical associations which relate fault manifestations to possible causes. For diagnosing *novel* faults the knowledge base contains models of implementation alternatives and execution behavior of functions indigenous to the domain. Both knowledge sources are utilized by a set of localization tactics to generate, select and test fault hypotheses.

II. KNOWLEDGE OF PROGRAM STRUCTURE AND FUNCTION

Knowledge of program structure and function is necessary for diagnosis of both known and novel faults. In the former case it is used to confirm expectations associated with a given fault hypothesis; in the latter case it is used to trace violations of expected behavior to their source. In our model this knowledge is captured in functional *phototypes*.

A functional. *phototype,* consists of four components: a set of recognition triggers, a list of pre- and post-conditions describing the execution behavior, a description of prototype components and their topology, and a List of constraints among components that must hold for recognition. Prototypes are defined at three levels of abstraction: the language, programming, and applications levels; and are hierarchically organized. A portion of the functional prototype hierarchy describing the component topology of the "master file priming read" (MFPR) function is given in Figure 1.

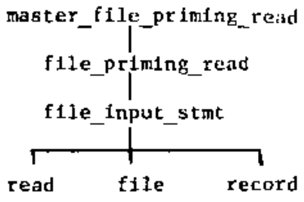

Figure 1. Topological Hierarchy for Master File Priming Read Functional Prototype

The figure shows that the MFPR consists of a single component, the file priming read. A file_priming read is only recognized as a MFPR if two constraints are met: the input record is used as a master record and the input file is used as a master file. Recognition triggers include the file priming_read and file input stmt prototypes as well as the language keyword "read".

III. KNOWLEDGE OF FAULTS

While functional prototypes describe expected program structure and function; fault *modals* describe expected defects in program structure and function. A fault *model* embodies knowledge of a particular implementation error or class of errors. This knowledge consists of conditions under which the model is applicable, a set of fault hypotheses, and a set of related fault models. A fault hypothesis is an expected defect in a functional prototype. Defects are defined as missing prototype components or violated constraints. Like functional prototypes fault models are hierarchically organized. Figure 2 details a fault model for a master file input error.

```
(master_file_priming_read error
   (triggered_by
      1.  invalid first master record
      2.  invalid first new master record
      3.  invalid master record count)
   (expected_defects
      1.  missing MFPR)
   (related_faults
      1.  insert_error)
```

Figure 2. Sample Fault Model

IV. LOCALIZATION TACTICS

Localization is performed using three tactics: *fault-driven, function-driven* and computation-*dntvan. Vault-driven.n localization* directs search at finding a particular kind of fault, such as an unexpected end_of file. *Function-driven localizatA.on* analyzes a particular functional prototype for an error. *Computation-driven localization* concentrates search on the computation of a set of program variables. Each makes use of an abstraction mechanism (Section 5) to generate and test fault hypotheses.

Multiple localization tactics are desirable for three reasons. First, they enhance the probability of finding faults since different tactics analyze the source code from different perspectives. Secondly, tactics which best fit the available information can be chosen. Thirdly, each tactic is optimal, in the sense of minimizing computational resources, for a particular localization task.

Fault-driven localization uses the fault model hierarchy to generate fault hypotheses. Output discrepancies serve as triggers for selecting a particular fault model. The more salient the trigger set the more specific the fault model and the associated fault hypotheses. Given a fault model, localization continues by first choosing one of these fault hypotheses and then invoking the abstraction mechanism to test it. If the hypothesis fails then a related hypothesis can be proposed, a different fault model can be applied, or the tactic can be abandoned.

Function-driven localization uses output discrepancies to suggest that a particular functional prototype is improperly implemented. The abstraction mechanism is then invoked to identify the code which implements the prototype. Tf the prototype cannot be found then the source code producing the most feasible match is considered as the intended implementation. The fault is identified by the statement causing failure in the matching process. This tactic can also be directed to examine a particular abstraction level for a fault. All prototypes defined at the chosen level are considered suspect. This application of the function-driven tactic is less directed and usually less desirable. Only when the output discrepancies offer no basis for prototype selection does this alternative become attractive.

Computation-driven localization traces the computation of a particular set of program variables. The computation is analyzed by extracting from the program those statements that directly affect the values of variables in the set. This extraction process is known as slicing [15]. The computation can be followed in either a forward or reverse order of execution flow. Faults are found by using the abstraction mechanism to interpret statements in the slice as members of more abstract functional prototypes and comparing the expected computational results to those derived.

V. PROGRAM ABSTRACTION

Program abstraction is performed by matching functional prototypes to the source code. Abstraction may either be expectation-driven or data-driven depending upon the localization tactic selected. Expectation-driven abstraction matches prototypes to code in a top-down manner, recursively matching components until a direct match can be made against the source code or previously recognized prototypes. Data-driven abstraction employs language-level triggers to select prototypes for matching. Prototypes matched at lower levels of abstraction serve as triggers for matching at higher levels. Once a functional prototype is identified the corresponding code is bound to it.

Recognition-based abstraction is an efficient technique, but it has limitations. Recognition of a functional prototype may fail for one of three reasons: a defect exists in the source code implementing the prototype, the wrong prototype is selected, or the source code represents an unfamiliar but correct implementation. The effectiveness of the recognition mechanism depends on the exactness of the triggering process and the richness of the alternative implementation set. Purely structural matching is augmented by functional matching. The behavior of a program segment which is inferred from language semantics, can also be compared to the functional descriptions in the prototype hierarchy.

VI. AN EXAMPLE

We have implemented an initial version of the model in a program named FALOSY (FAult Localization SYstem). FALOSY addresses faults in master file update programs [16]. In this section we illustrate FALOSY's reasoning for the master file priming read error.

FALOSY is presented with a discrepancy list and a list representation of the program's source code. The discrepancy list formally describes differences between expected and observed output. An abridged trace of FALOSY's reasoning is given in the Appendix. Numbers in parentheses refer to line numbers in the Appendix.

A production system, whose antecedents are sets of discrepancies, is used to select the initial localization tactic. In this case the fault-driven tactic is chosen and the master file priming read fault model is triggered (1). FALOSY hypothesizes that the priming, read for the master file is missing (2). The abstraction mechanism is invoked to identify the corresponding prototype (4). A check is first made to determine if it has been previously identified. Since it has not, the recognition mechanism is invoked recursively to find the components oi the master file priming read prototype.

Eventually search is carried out at the source level, and three read statements (6, 26) are selected for further matching. Constraints are now checked starting with those at the lowest level in the abstraction hierarchy. The first candidate is rejected since the file identifier is not used as the master file (20). The second and third candidates are rejectee because they do not precede the update loop (24,28). No candidate satisfies all constraints and recognition fails. The original hypothesis is thus verified.

APPENDIX

```
1. Applicable fault models (m_f_pr)
2. Expected defects  ((m_f_pr missing))
3. Fault hypothesis (m_f_pr missing))
4. Trying to recognize: m_f_pr
5. Triggers are (read f_i_s_ f_p_r
6. Matching m_f_pr to (s6 read transfile transbuf)
7. m_f_pr has role f_p_r
8. Matching f_p_r to (s6 read transfile transbuf)
9. f_p_r has role f_i_s
10. Matching f_i_s to (s6 read transfile transbuf)
11. f_i_s is primitive
12. Checking constraint (data-type file pascal_fid)
13. Checking constraint (data-type record pascal_rec)
14. Match for f_i_s succeeds
15. Checking constraint (before f_i_s update_loop)
16. Trying to recognize:  update-loop
17. Recognition of update-loop succeeds
18. Match for f_p_r succeeds
19. Checking constraint (use file m_f)
20. Role file, which is bound to transfile, is used
    as a t_f whereas m_f was expected
21. Match for m_f_pr fails
22. Matching m_f_pr to (s24 read oldfile oldbuf)
23. Checking constraint (before f_i_s update-loop)
24. Hole f_i_s, which is bound to g 00014, does not
    precede role update loop, which is bound to
    g0005
25. Match for f p r fails
26. Matching m f pr to (s52 read transfile transbuf)
27. Checking constraint (before f_i_s update loop)
28. Role f i s, which is bound to gOOO18, does not
    precede role update loop which is bound to
    g()0005
29. Match for f_p_r fails
30. Recognition of m f pr fails
31. Fault hypothesis verified
32. Fault is: (m f pr missing)
```

BIBLIOGRAPHY

[11 J. DeKleer, "Local methods for localization of faults in electonic circuits," M1T-A! Memo 394, 1976.

[2] M.Genesereth, "Diagnosis using hierarchic design models," *Proceeding* of the Second* AAA] *Conference-,* August, 1982.

[3] R. Davis, H. Shrobe, w. Hamscher, K. Wieckert, M. Shirley, and S. Pol it, "Diagnosis based on description of structure and function," proceeding of *the Second* AAA! *Coference,* August, 1982.

[4] R. Hartlev, "How expert should an expert system be?", *Proceeding* I*JCAI-81, August 1981.

[5] R. Patil, P. Szolovits, W. Schwartz, "Causal understanding of patient illness in medical diagnosis," *Proceedings of 1JCA1-81,* August, 1981.

[6] E. Short life, *Computer Based Medical Consultations' MYCIN,* American Elsevier, New York, New York, 1976.

[71 B. Chandrasekaran and S. Mittal, "Deep versus compiled knowledge approaches to diagnostic problem-solving," *Proceeding* of the Second* AAAI *Conference,* August, 1982.

[81 H. Wertz, "Understanding and improving LISP programs," 7 TCAI-77 *Proceeding*.*

[91 R. Ruth, "Intelligent program analysis," artifical *Intelligence,* Vol. 7, No. I, 1976

[101 C. Adam and M. "LAURA, a system to debug student programs," *Artificial Intelligence,* November, 1980.

[11] F. "Understanding and debugging computer programs," International *Journal of Man-Machine. Studies,* Vol. 12, 19S0, pp. 189-202.

[12] G Sussman, A *Computational Model of Skill Acquisition,* American Elsevier, New York, 1975.

[13]Miller, "A structured planning and debugging environment," *International Journal of Man-Machine Studies*, Vol. 11. 1978.

[14]D. Shapiro, "Sniffer: a system that understands bug," MIT-AI Memo 638, June, 1981

[151M. Weiser, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method," University of Michigan Ph.D. Thesis, 1979.

[16]B. Dwyer, "One more time - How to update a master file," *Communications of the ACM,* Vol. 24, No. 1, January 1981.