NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

PROGRAM SYNTHESIS BY INDUCTIVE INFERENCE

Matthew M. Huntbach

1986

CABINET

Cognitive Science Research Paper

Serial No. CSRP 059

Cognitive Studies Programme,
School of Social Sciences,
The University of Sussex
Brighton BN1 9QN

Cognitive Studies Programme
University of Sussex
Brighton, BN1 9QN
England.

1. INTRODUCTION

In his thesis 'Algorithmic Program Debugging' [16], E.Y.Shapiro presents a theoretical framework for program debugging using an inductive inference mechanism. The general nature of this framework means that, by specifying an initial empty program, it may also be used for the inference of programs from examples of input/output behaviour.

We have implemented Shapiro's Model Inference System, and made a number of changes and additions to it. In particular, we no longer regard it as specifically a tool for Prolog programmers, but as a general program synthesis system. Our efforts have been concentrated on producing a practical refinement operator [17].

This paper describes our system without using logic programming terminology. We also indicate how we see this system becoming part of a collection of tools to provide automated assistance to the computer programmer.

2. DATA FLOW PROGRAMS

Shapiro's framework adapts most easily to the logic programming language Prolog, and his system uses Prolog as both the implementation and target language. The work presented here relies on Shapiro's framework, and does in fact use a subset of Prolog as the target language. But we find it easiest to think of our target programs in the form of generalised data flow diagrams [4]. We are influenced particularly by the plan notation of the Programmer's Apprentice [19]. Our system is implemented in a combination of Prolog and Pop-11, using the Poplog system [13].

In our notation, a procedure is shown by a set of data flow diagrams, each representing one particular path through the program. All diagrams for a particular procedure have the same number of input and output ports representing that procedure's parameters. A diagram consists of a number of segments which themselves have input and output ports, linked by data flow lines. A procedure may be incomplete, in which case there are possible paths through the program for which no diagram is present.

The segments represent calls to subprocedures or tests. A procedure segment when presented with input values or tokens produces output tokens which are assumed to flow along the data flow lines. A test produces no output tokens but instead either succeeds or fails. If any test fails, the control path represented by the diagram is the wrong one for the given input, and another is tried, until one succeeds in which case the tokens which arrive at the output ports of the diagram are the output of the whole procedure. The program fails if no program path

possible for that procedure to receive. The tests and
subprocedures may themselves be represented by further data flo
diagrams, they may be recursive calls, or they may be system
primitives. Any program path on which a subprocedure fails also
fails.

A program path is said to 'cover' a given input/output pair if,
when presented with the given input, it succeeds and produces the
given output when the execution of the program path is simulated
by executing only system primitives and asking the user for the
output he would expect from the other subtests and subprocedure
We term this 'query execution'. It is equivalent to Shapiro's
definition of 'cover' using the 'eager strategy' – we do not
consider his other search strategies here.

Figure 1 gives an example for a program which inserts an integer
in position into an ordered list of integers (the program is in
fact incomplete and contains bugs). The system primitives are
"head", "tail", "isnil", ">", "cons", and "createnil" which takes
no inputs and always returns the empty list as output. If "t" is a
test we also have "not(t)" which succeeds when t fails and vice
versa. For convenience we will assume in future that "head" and
"tail" have a built-in test for "not(isnil)".

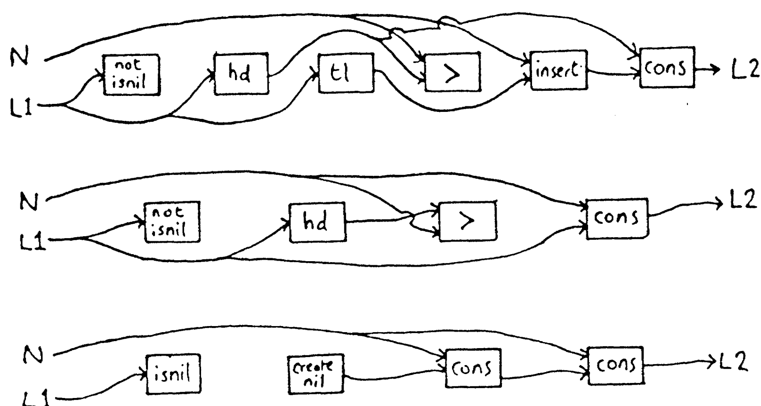

Figure 1

To test whether the first program path covers the pair
insert(2,[1,2,3]) -> [1,2,2,3] the program path would be executed
until it came to the recursive call 'insert(2,[2,3])'. The user
would then be asked for the result. He would enter [2,2,3] (this
is the expected result rather than the one that would be produce
were the program fully executed) and the execution would continue,
giving the output [1,2,2,3], so the input/output pair is covered
by the path.

3. PROGRAM DIAGNOSIS

Shapiro presents three algorithms to diagnose faults in three
different cases where bugs are found. Each of these will
interactively query the user during execution.

program with the input and expected output. It returns an input/output pair which the user defines as true but which is not covered by any existing path in the program.

For example, the program in figure 1 will fail when called with input insert(2,[1,2,3]), expected output [1,2,2,3], because there is no program path to cover the case of inserting an integer into a list whose head is equal to the integer. The algorithm will return insert(2,[2,3]) -> [2,2,3] as the uncovered input/output pair.

Algorithm 2 is used when the program succeeds on a given input, but the output produced is different from that expected. It takes the program with the input and incorrect output produced. It returns a program path which covers a false input/output pair.

For example, the program in figure 1 if called with input insert(4,[1,2,3]) will return [1,2,3,4,4] rather than the expected [1,2,3,4]. The algorithm will find that the third program path covers the input output pair insert(4,[]) -> [4,4] which the user indicates should be false.

Algorithm 3 is used when the program fails to terminate. It returns a program path which is looping. It works by keeping a stack of procedure calls to a fixed limit. When this limit is exceeded, the program is assumed to be looping. The stack is searched to find any point where a procedure with a given input calls itself with the same input, or where some violation of a well-founded ordering on procedure calls occurs [5]. The algorithm returns the program path responsible for the violation.

4. PROGRAM SYNTHESIS

Shapiro's diagnosis algorithms have been fairly widely implemented, not only for Prolog, but also for other languages such as Pascal [14]. Less attention has been given to his use of these algorithms in inductive program synthesis.

The basis of this system is that a series of input/output examples is presented by the user. Each of these is tested in turn against the current conjectured program. If an example indicated as true fails, or one indicated as false succeeds, the diagnosis algorithms are used to detect a bug, and the conjectured program is replaced by one with this bug corrected which also behaves correctly with respect to all other input/output examples presented previously. If the conjectured programs are restricted to be any recursively enumerable class of programs that are everywhere terminating then, using concepts introduced by Gold [7], the correct program may be identified in the limit.

The inference algorithm is incremental, that is any path which was removed for covering a false input/output example is never added again to the program. The complete algorithm is shown in figure 2.

The diagnosis algorithms generate a considerable number of queries, the answers to which are added to the list of known input/output pairs. Whenever a query is made, this list is first searched to check whether the answer has been given previously, thus cutting the number of queries made directly to the user.

```
repeat
   read the next input/output pair
   repeat
      if the program P fails on a true example then
         use Algorithm 1 to find an uncovered input/output
            pair A
         search for an unmarked program path p that covers I
         add p to P
      endif;
      if the progrmm P succeeds on a false example or
         produces an output different from that expected
      then
         use Algorithm 2 to detect a false program path p
         remove p from P, And mdd it to the set of marked
         program paths
      endif;
   until P is correct with respect to all known input/output
            pairs;
until no input/output pairs are left to read;

interrupt if the depth of a computation on some input/output
            pair A exceeds some limit
            use Algorithm 3 to find a program path p which
            is looping
            remove p from P, add it to the set of marked
            program paths
            restart the computation of A
endinterrupt
```

Figure 2

5. SEARCHING FOR NEW PROGRAM PATHS

rhe major part of our research has been to improve the method u to search for a new program path to cover a given input/output ?air. In an earlier paper, we describe Shapiro's method with "espect to Prolog in detail, and suggest some improvements C103. Here, we give a simple method of searching the space of possible program paths using our data flow notation. Our simplifications are due mainly to our use of separate "hd", "tl" and "cons" operations, rather than the list templates of Prolog clauses.

4e introduce the concept of an incomplete program path. This is a program path in which there are no data flow lines leading to the Dutput ports of the main procedure. In the search tree for covering program paths, the non-leaf nodes *Are* incomplete program :>aths, the leaves *Arm* complete. The root node is an incomplete program path consisting of Just the input and output ports with ri iata flow lines at all.

fhere *Arm* three refinement operations by which children may be generated:

L) Add a subprocedure. This involves constructing data flow lines either from the input ports of the main procedure or the outp ports of existing subprocedures. The output ports of the new subprocedure *Arm* unconnected.

constructing aaia now lines to tne output port» OT tne main procedure from either its input ports or the output ports of tubproccdures.

lo subprocedure may be added that exactly duplicates the main procedure (as this would lead to a looping path) or exactly kiplicates a subprocedure already added. No program path may be rlosed in such a way that there exists a subprocedure which has lataflow lines from any of its output ports (since such a iubprocedure would be superfluous).

:or each procedure used, the user must declare
) The number of input and output ports it has, and the type of the data items each port uses.
I) The subprocedures and tests it may call.

:or example, the insert procedure of figure 1 is declared to have :wo input ports: N type integer, LI type list(integer) and one Hitput port: L2 type list(integer). It is declared to make use of >ne test ">" and one subprocedure, "insert" (that is, itself :alled recursively). In addition it is assumed that each procedure lay make use of various system procedures and tests such as sir ist operations.

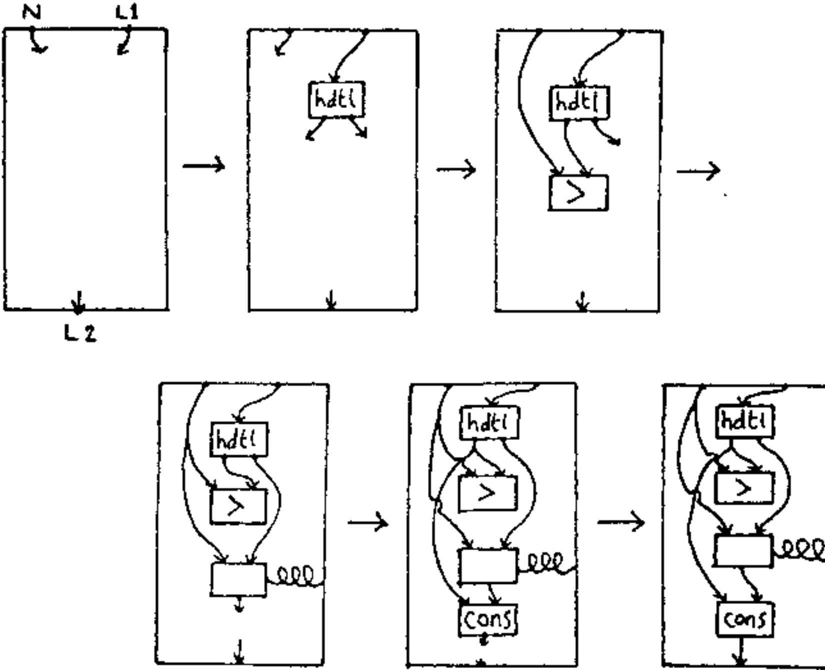

Figure 3

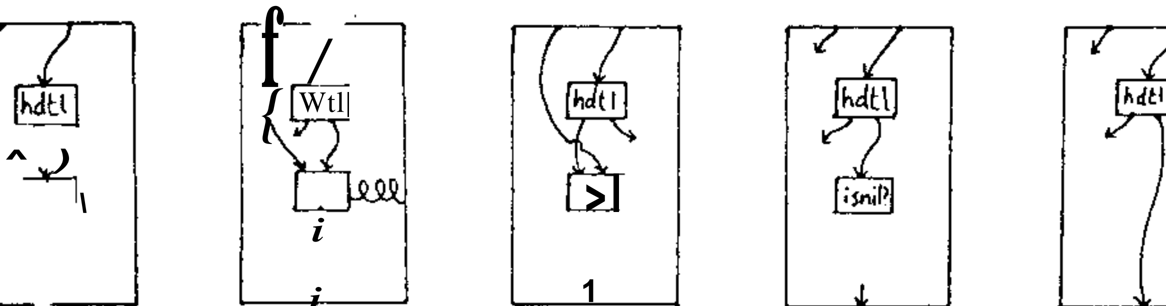

Figure 4

call uy a coiled line. r lyur e 4 shows some of the possible siblings
Df the second node of figure 3.

Heuristic values may be assigned to program paths to guide the search, though we do not consider this further here. When a complete path is found during the search, it is tested to see whether it covers the input/output pair. For example, if we *mrm* searching for a path to cover insert(3,Cl,2,43) -> Cl,2,3,43 the fifth path in figure 4 does not cover it since it leads to Lnsert(3,Cl,2,43) -> C2,4]. An incomplete path may also be tested to see if it covers a pair. It fails to cover it if it contains a test which fails. The third *4knd* fourth paths in figure 4, for example, fail to cover insert<3,Cl,2,43) -> Cl,2,3,43 since they Lead to the tests 1>3 and isnil(C2,4]> respectively. Any descendant of an incomplete path which fails to cover a pair will also fail to cover it, so it need not be searched and the tree c« >e pruned. A complete covering path may be found which is, lowever, marked, that is it has already been removed from the Drogram because it leads to false output for another input. In this case, search continues for an alternative covering path.

b. GOAL DIRECTED CLOSURE

Consider the program path shown in figure 5. This might be generated during the search for a program path to cover the pair dbl2nd(Ca,b,c,d,e,«) -> Ca,b,b,c,d,d,e,f,«, dbl2nd being a procedure to double every second element in a list. If this Incomplete path were query executed (resulting in the user being queried for the result of dbl2nd(Cc,d,e,fJ», the values shown iiould be produced at each output port.

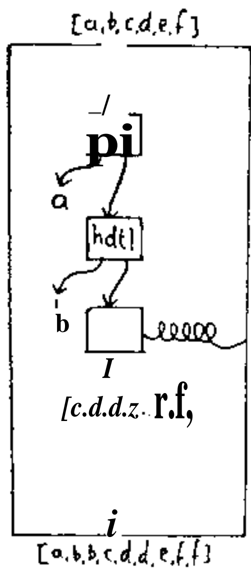

Figure 5

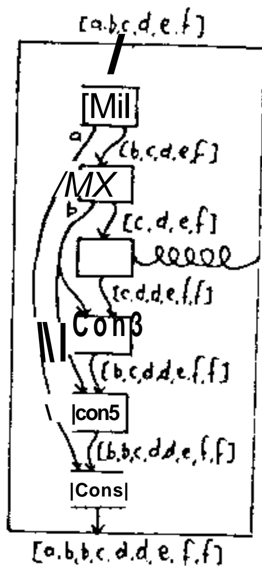

Figure 6

: or more (the system would generate separate paths to cover th
base cases of the empty list or a list of one element).

We add to our search algorithm an operation which query executes each incomplete program path using the input for the pair which needs to be covered. The values produced at each of the output ports are stored, and a test is made to see if these may be use to construct the output for the pair and thus produce a coverinc path. If there is more than one way of constructing the output, each is produced in case some are marked paths.

Using this technique, heuristic search of program paths produced by adding construction operations and closure is unnecessary.

7. CREATING AUXILIARY PROCEDURES

If, while testing for a goal directed closure, it is found that it is possible to create only part of an output value, the program path may be closed by creating an auxiliary procedure which produces the rest. Figure 7 shows such a closure for the procedu "combs" which takes a list as input and produces as output a list of all two-element combinations of items in the list.

A program path to cover the new pair aux([a,b,c,d]) -> [a,b],[a,c],[a,d]] must now be synthesised. However, the user has not specified the use of the procedure "aux" and so cannot be expected to answer queries on it (as would be necessary if query executing a recursive path in the search for a covering path). It is however possible to use a backwards reasoning to give the answer to queries involving auxiliary procedures. In the case of Figure 7, the answer to any query involving the procedure "aux" may be found by applying the same input values to the path in Figure 8 and query executing it. The system procedure "bsub" take the seconds argument from the back of the first, that is if append(A,B) = C then bsub(C,B) = A.

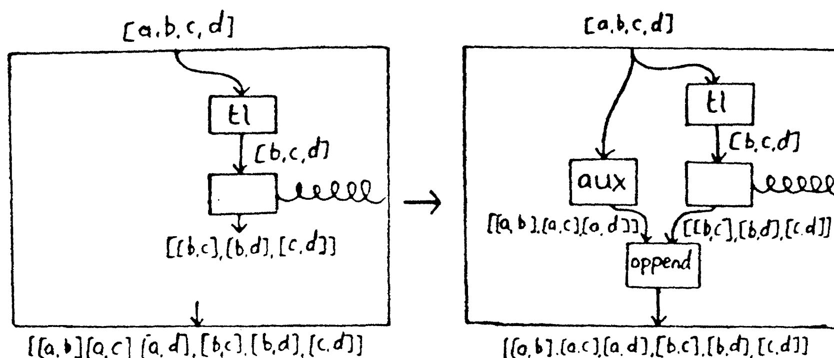

Figure 7

So to find the output of aux([a,b,c]) the user would be asked first for the output of combs([a,b,c]) then for the output of combs([b,c]) that is [[a,b],[a,c],[b,c]] and [[b,c]] respectively.

not already a member of that list, otherwise it returns the list unchanged. Then if we present the system with the pair addnew<e,£a,b,c9d3) —> £e9a9b9c,dl it will produce the program path shown in -figure 9 to cover it.

However, this program path will be removed when the system is presented with the pair addnew(c,Ca9b9c,dl) —> Ca,b,c,d] since it leads to the false input/output pair addnew(c,r.a,b,cfd3> —> r.c9ayb9c,dL The system then needs to search for a new path which covers addnew(e9Ca,b9c,d3) -> Ce9a9b9c,d3 but not addnew<c9Ca,b,c,d3) -> Cc9a,b,c,d3.

One possibility is to modify the removed path by adding an auxiliary test as in figure 10. The two input/output pairs auxtest<e,Ca9b,c,d3> -> true and auxtest(c,r.a,b,c,d3) -> false are added to the databse of pairs, and the system will search for a program to cover them.

For any queries of auxtest<X9Y) the user is asked for the result of addnew(X9Y). auxtest(X,Y) is defined as true when addnew(X9Y) « cons(X,Y), false otherwise. This may be used to sythesise "auxtest[11] (in fact9 our system would identify it as being equivalent to the system test "not(member)[11]).

Although the generation of auxiliary procedures and tests may le< to inefficient or unusually structured programs, we envisage program transformation techniques C33 being applied to them. We have already implemented a simple transformation which incorporates the program path to cover some auxiliary into the parent path if the program for the auxiliary is found to be a single path.

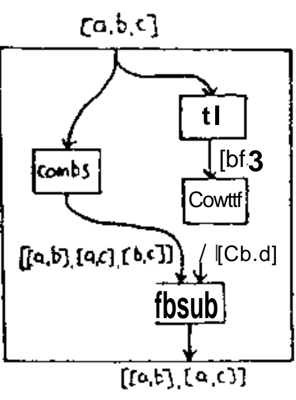

Figure 8

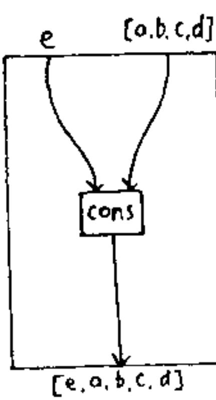

Figure 9

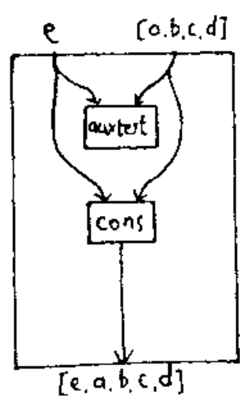

Figure 10

8. PROGRAM PATHS TO PROGRAMS

A set of program paths may be represented and executed as a set Prolog clauses by representing each path as a clause, and each subprocedure and test segment in a path as a subgoal. Each outp« port is represented by a new Prolog variable. A data flow line is

owever, as we have indicated, our intention is that each program ath should be thought of as one particular flow of control hrough a program. Our program paths may be combined to form plar iagrams as used by the M.I.T. Programmer's Apprentice [15]. This lan notation is claimed to represent all the essential details of program while suppressing those incidental features which stem rom particular programming languages. Translators to and from the lan notation and several standard programming languages have bee ritten. Thus, with the appropiate translator, our system could be sed to synthesise programs in any standard language.

e envisage our program synthesis system being incorporated as o lement in a larger programmer's assistant type system such as ha een suggested with PSI [8]. We agree with Kant [12] that ncorporating an inductive inference capability into a program ynthesis system makes sense; expecting it to solve the entire rogram synthesis problem does not'. It may best be used perhaps or filling in gaps or correcting mistakes that inevitable arise hen programs are methodically derived from specifications [6] specially from informal specifications [1].

. CONCLUSIONS

ur work represents a further step from theoretical work on nductive inference [7],[2] towards a tool which may be of ractical use to computer programmers. The link is Shapiro's Model nference System [16]. We have demonstrated that there is great cope for improvemnets to be made to this system, particulalrly t he program synthesis part.

e have also tried to separate Shapiro's work from the field of ogic programming. Although the use of Prolog was important in llowing the construction of a demonstratable system, we feel the ork needs to be seen more in its own right.

n the field of program synthesis from examples, we have produced system which can cope with most of the examples dealt with by ther systems [9], [18], [11] as well as some not previously ynthesised entirely from examples.

0. ACKNOWLEDGMENTS

inancial assistance for this work was given by the Artificial ntelligence group at the G.E.C. Marconi Research Centre. I also hank my supervisor, Rudi Lutz, and other people in the Cognitive tudies Programme at Sussex University for helpful comments.

1. REFERENCES

[1] R.Balzer, N.Goldman, D.Wile. Informality in program specification. IEEE Trans. Soft. Eng. SE-4,2 pp.94-103. 1978.

[2] L.Blum & M.Blum. Towards a mathematical theory of inductive inference. Information and Control 28. 1975.

[3] J.Darlington. Program transformation. In 'Functional Programming and its Applications', pp.193-215. Eds. J.Darlington, P.Henderson, D.A.Turner. Cambridge Univ. Press, U.K. 1982.

[4] J.B.Dennis.Dataflow supercomputers. IEEE Computer 13,11 pp.48-56. 1980.
[5] R.W.Floyd. Assigning meaning to programs. In Proc. Symp. on App. Math. Ed. J.D.Schwartz. American Math. Soc. 1967.
[6] S.L.Gerhart & L.Yelowitz. Observations of fallibility in applications of modern programming methodology. IEEE Trans. Soft. Eng. SE-2,3 pp.195-207. 1976.
[7] E.M.Gold. Language identification in the limit. Information and Control 10 pp.447-474. 1967.
[8] C.Green. A summary of the PSI program synthesis system. IJCAI-77 pp.380-381. 1977.
[9] S.Hardy. Synthesis of Lisp programs from examples. IJCAI-75 pp.240-245. 1975.
[10] M.M.Huntbach. An improved version of Shapiro's Model Inference System. Third International Logic Programming Conference, London 1986.
[11] J.P.Jouannaud and G.Guiho. Inference of functions with an interactive system. Machine Intelligence 9, pp.227-250. 1979.
[12] E.Kant. Understanding and automating algorithm design. IEEE Trans. Soft. Eng. SE-11, 11. pp.1361-1374. 1985.
[13] C.Mellish & S.Hardy. Integrating Prolog into the Poplog environment. In 'Implementations of Prolog', ed. J.A.Campbell. Ellis Horwood 1984.
[14] S.Renner. Location of logical errors in Pascal programs with an appendix on implementation problems in Waterloo Prolog/C. Internal report UIUCDCS-F-82-896. Knowledge Based Programmer Assistant Project, University of Illinois, 1982.
[15] C.Rich. A formal representation for plans in the Programmer's Apprentice. IJCAI-81 pp.1044-1052. 1981.
[16] E.Y.Shapiro. Algorithmic Program Debugging. M.I.T. Press 1982.
[17] E.Y.Shapiro. Inductive Inference of Theories from Facts. Yale Univ. Dept. of Computer Science. Research Report 192. 1981.
[18] D.E.Shaw, W.R.Swartout, C.Green. Inferring Lisp programs from examples. IJCAI-75, pp.260-267. 1975.
[19] R.C.Waters. The Programmer's Apprentice: knowledge based program editing. IEEE Trans. Soft. Eng. SE-8,1 pp.1-13. 1982.