

THE ROLES OF KNOWLEDGE AND DEDUCTION IN PROGRAM SYNTHESIS

David R. Barstow
Department of Computer Science
Yale University
P.O. Box 2158 Yale Station
New Haven, CT 06520

In the earliest attempts to apply artificial intelligence techniques to program synthesis, deduction (that is, the use of a general purpose mechanism such as a theorem prover) played a central role. Some recent systems have relied almost exclusively on knowledge about programming in particular domains, with no significant role for deduction. Even in such knowledge-based systems, however, there seems to be an important role for deduction in testing the applicability conditions of specific programming rules. This auxiliary role for deduction can be seen clearly in a hypothetical synthesis of a breadth-first enumeration algorithm. The hypothetical synthesis also demonstrates the utility of detailed programming knowledge for dealing with non-algorithmic specifications.

1. INTRODUCTION

In the earliest attempts to apply artificial intelligence techniques to the programming problem, deduction (i.e., the use of a relatively general purpose deductive mechanism such as a theorem prover) played a central role. The Heuristic Compiler [9], based on the GPS formalism, used means-end analysis to determine a sequence of operators that achieved the desired result. Green [4] and Waldinger [10] developed systems that used resolution theorem provers to construct programs. First, a proof was developed for a theorem derived from the input/output specifications of the program. Then a program was constructed directly from the theorem's proof. The only knowledge (i.e., specific detailed facts, rules, or procedures) available to these early systems was knowledge about the target language. In the Heuristic Compiler, this was encoded in the operator difference tables. The systems of Green and Waldinger had axioms describing various target language constructs. Later, Manna and Waldinger [6] showed the importance of knowledge about the domain, in addition to knowledge about the target language. Their recent work [7] has also emphasized the utility of knowledge about

This material is based upon work supported by the National Science Foundation under Grant No. MCS78-03827. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

certain general aspects of programming, such as the introduction of conditionals and recursive calls, but still within a deductive framework.

In contrast, PECOS [2, 3] represented an application of a different artificial intelligence paradigm, knowledge engineering, to the programming problem. PECOS had a knowledge base of about four hundred specific rules (or facts) about many aspects of symbolic programming, including such abstract concepts as sorting, enumeration, collections, and mappings, and such concrete concepts as loops, lists, arrays, property lists, and record structures. PECOS could apply these rules to the task of implementing abstract algorithms. PECOS was able to implement algorithms in a variety of domains, including symbolic programming (simple classification and concept formation algorithms), graph theory (a reachability algorithm), and simple number theory (a prime number algorithm). PECOS's knowledge of alternative implementation techniques enabled the construction of a variety of implementations for each algorithm.

Although deduction (in the sense described above) played no significant role in PECOS, there is a rather important role that deduction can and should play in knowledge-based automatic programming systems. This role can be seen by considering a relatively simple rule:

If you know that an object is larger than (or equal to) every member of an ordered sequence, the object may be added to the sequence by inserting it at the back.

This rule is something that human programmers probably know, and seems reasonable to include in an automatic programming system. The question is: How should the condition in the rule be tested? There are many and varied situations in which the condition holds. One simple case is when the insertion routine for a selection sort is being written; another will be seen in the next section. Since there are so many situations in which the condition holds, the right way to test it seems to be to pass it off to some mechanism that can test whether or not it holds for the program being constructed -- that is, to a relatively general deductive mechanism that can test and prove properties about programs (and partially developed ones). Thus, we see that deduction plays an important (if auxiliary) role when retrieving and applying this rule, and it is this role that will be the focus of the remainder of this discussion. This is not to say that other roles for deduction are unimportant for automatic programming, but merely that this role is especially important for knowledge-based automatic programming systems, and is one that has received relatively little emphasis in automatic programming research. (The one notable exception is the Programmer's Apprentice [8].)

In the next section, a hypothetical synthesis of a graph theory algorithm will be presented. The synthesis consists of a sequence of reasoning steps, each suggesting a fact or rule that seems useful for an automatic programming system to know. Several of the steps depend on some condition holding for the program under construction. These are the conditions that seem appropriate tasks for a deductive system, and are indicated by the word "PROOF" in the synthesis. Note that the synthesis is driven primarily by the programming knowledge, rather than by the deductive system. The synthesis also demonstrates the utility of programming knowledge for non-algorithmic program specifications. (Recall that programs were specified to PECOS as abstract algorithms.) Finally, it should also be noted that the problem considered in the next section is beyond the capabilities of any current automatic programming system.

2. HYPOTHETICAL SYNTHESIS

2.1 Specification

Write a program that inputs the root R of a tree with a known successor function $CHILDREN(N)$ and constructs a sequence of all (and only) the nodes of the tree, such that if the distance from R to N_1 is less than the

distance from R to N_2 , then N_1 precedes N_2 in the sequence (where the distance from X to Y , denoted by $D(X,Y)$, is the number of arcs between X and Y ; thus, $D(R,X)$ is the depth of X).

This specification is basically a non-algorithmic description of a breadth-first enumeration of the nodes in a tree. One way for an automatic programming system to deal with this specification would be to recognize that a breadth-first enumeration is the desired program, and to use specific knowledge about breadth-first enumerations (perhaps even a template) to produce a program. In this hypothetical synthesis, we will consider the more interesting case in which the basic algorithm must actually be developed.

2.2 Overview

The synthesis process goes through four major stages. First, the task is broken into two parts, a "producer" that generates the nodes in the desired order, and a "consumer" that builds a sequence from the nodes generated by the producer. The second stage involves constructing the consumer, which simplifies into a simple concatenation operation. The third stage, in which the producer is built, is relatively complex but eventually a simple queue mechanism is developed, in which nodes are generated by taking them from the front and the children of each generated node are added at the back. Finally, the two processes are combined together into a simple WHILE loop.

2.3 The Synthesis

(1) Since the input is a tree and the output is a sequence whose elements are the nodes of the tree, the central task is one of using objects from one structure (the set of nodes) to build another structure (the sequence of nodes). Such a task can be implemented by building two processes, a producer and a consumer. The producer produces the objects one at a time and the consumer consumes them. In this case, the producer must produce all of the nodes of the tree and the consumer must build the desired sequence from them.

(2) When objects are being transferred one at a time, there is some order in which the objects are being transferred. This temporal order will be referred to as the "transfer order." There are many possible transfer orders. When the task of the consumer is to build a sequence satisfying some ordering constraint, one possibility is to force the transfer order to

satisfy the same constraint. In this case, the constraint is:

if $D(R,X) < D(R,Y)$ then transfer X before Y

(3) The consumer is a process that accepts objects one at a time and builds a sequence S from them. The build process is one of adding an individual object to S. After each addition, S must satisfy a particular constraint:

if $D(R,X) < D(R,Y)$ then X precedes Y in S

(4) Initially, S must be empty. Thus, the initialization step of the process is the following:

S := <>

where <> denotes the empty sequence.

(5) For sequences, the addition process involves finding a suitable position to add the new object, followed by inserting it there. In this case, since P follows every element in S according to the ordering constraint (proof given below), the back is always a suitable position. Thus, the addition step (the body of the consumer) is:

P := receive from producer
insert P at the back of S

PROOF: P follows every element in S

The ordering constraint on S is:

if $D(R,X) < D(R,Y)$ then X precedes Y in S

The order in which the consumer receives the objects is:

if $D(R,X) < D(R,Y)$ then X is received before Y

Thus, if X must precede P in S, X is received before P. Thus, when P is received, any object that must precede P in S has already been received. Therefore, P follows every element in S. QED

(6) The consumer is finished:

Initialization:

S := <>

Body:

P := receive from producer
S := S . <P>

where "." signifies concatenation of sequences.

(7) The producer is a process that produces objects one at a time, guaranteeing that each object is produced once and only once. The set of objects to be produced is the set of nodes of the tree. The order in which they are produced (the "production order") must satisfy the following constraint:

if $D(R,X) < D(R,Y)$ then produce X before Y

(8) In a tree, the set of nodes is the set of objects that are reachable from the root of the tree by following zero or more arcs. How should this set be constructed? The obvious way is to follow all arcs from the root as far as possible.

(9) At any point in the process of following all arcs, there will be certain nodes that have been found and certain that have not. Of those that have been found, some will have had their successors looked at and others will not. Thus, at each point, the state of knowledge about the nodes can be described as a mapping, which we will call MARK:

Domain: nodes

Range: {"CHECKED", "UNCHECKED", "UNKNOWN"}

with the following semantics:

MARK[X] = "CHECKED"

iff X is known to be reachable
and X's successors have been checked

MARK[X] = "UNCHECKED"

iff X is known to be reachable
and X's successors have not been checked

MARK[X] = "UNKNOWN"

iff X is not known to be reachable

(10) Initially, all that is known to be reachable is the root, and its successors haven't been checked; everything else is not known to be reachable:

MARK[X] = "UNCHECKED" iff X is the root

MARK[X] = "UNKNOWN" iff X is not the root

(11) Each step in the process consists of picking one node that is known to be reachable but whose successors haven't been checked, and to check its successors (as well as marking it as being checked). If a successor is "UNKNOWN" then change it to "UNCHECKED". But if it is "CHECKED" or "UNCHECKED", then don't do anything with it, since it has already been found. Thus, the body of the process is:

X := any node marked "UNCHECKED"
 change MARK[X] from "UNCHECKED" to "CHECKED"
 for all successors, Y, of X:
 if MARK[Y] = "UNKNOWN"
 then change MARK[Y] from "UNKNOWN"
 to "UNCHECKED"

(12) When no "UNCHECKED" nodes remain, all of the arcs have been followed, and everything marked "CHECKED" is reachable. Hence, the termination test is:

is MARK-1["UNCHECKED"] empty

where MARK-1 denotes the inverse image under the MARK mapping. The set of reachable nodes is:

MARK-1["CHECKED"]

(13) We now have a way to determine the set of nodes to be produced. In order to actually construct a producer, we need a way of enumerating them. One way would be to construct the set and then enumerate the objects in the set. In this case, however, we can produce them as we go along. We can do this since each node is marked "CHECKED" only once.

PROOF: a node is marked "CHECKED" only once

Initially no nodes are marked "CHECKED", and there are only two changes that happen in the mapping:

change MARK[X] from "UNCHECKED" to "CHECKED"
 change MARK[X] from "UNKNOWN" to "UNCHECKED"

Thus, once a node is marked "CHECKED", it's marking is never changed. QED

(14) But this technique of producing them as they are checked will work only if the X's are picked to satisfy any ordering constraints on the producer. So the producer body is now the following:

X := any node in MARK-1["UNCHECKED"]
 change MARK[X] from "UNCHECKED" to "CHECKED"
 send X to consumer
 for all successors, Y, of X:
 if MARK[Y] = "UNKNOWN"
 then change MARK[Y] from "UNKNOWN"
 to "UNCHECKED"

where the "any" operation must satisfy the following constraint:

if $D(R, X) < D(R, Z)$ then X is selected before Z
 (even if X or Z is not marked "UNCHECKED")

(15) The constraint can be satisfied if the node X selected by the "any" operation satisfies the following property:

$D(R, X) \leq D(R, Z)$ for any Z marked "UNCHECKED"
 or marked "UNKNOWN" and reachable from R

(16) If the X selected is the closest of those nodes marked "UNCHECKED" or "UNKNOWN", then the property will be satisfied. This can only be done if one of the nodes marked "UNCHECKED" satisfies the property. If there is such a node, that node must be the closest of the nodes marked "UNCHECKED". However, how do we know that such an X is also closer than those nodes marked "UNKNOWN"?

PROOF: if X is the closest "UNCHECKED" node
 and Z is "UNKNOWN" and reachable from R,
 then $D(R, X) \leq D(R, Z)$

Since Z is reachable from R, there is some path from R to Z. Then Z is reachable from some node marked "UNCHECKED", call it W. This is true since either (i) R is "UNCHECKED" and W can be R; or (ii) the path from R to Z begins with some nonzero number of "CHECKED" nodes and the next node (which is the desired W) cannot be "UNKNOWN", since all of a node's "UNKNOWN" successors are changed to "UNCHECKED" as the node is changed from "UNCHECKED" to "CHECKED". Since $0 \leq D(W, Z)$, and since, in a tree, there is only one path from R to Z or W, $D(R, W) \leq D(R, Z)$. Since W is "UNCHECKED", $D(R, X) \leq D(R, W)$. Hence, $D(R, X) \leq D(R, Z)$. QED

(Note that a proof not dependent on the graph being a tree would have to use the fact that the closest node was being taken each time.)

(17) The producer's action has been simplified:

X := closest node marked "UNCHECKED"
 change MARK[X] from "UNCHECKED" to "CHECKED"
 send X to consumer
 for all successors, Y, of X:
 if MARK[Y] = "UNKNOWN"
 then change MARK[Y] from "UNKNOWN"
 to "UNCHECKED"

(18) Part of the body involves a test of whether a node is "UNKNOWN". We will now see that this test must necessarily return "TRUE", and hence can be omitted from the action of the producer:

X := closest node marked "UNCHECKED"
 change MARK[X] from "UNCHECKED" to "CHECKED"
 send X to consumer
 for all successors, Y, of X:
 change MARK[Y] from "UNKNOWN"
 to "UNCHECKED"

PROOF: every successor of X is "UNKNOWN"

Every successor of X was initially "UNKNOWN", since every node (except the root, which is not a successor of any node), was initially "UNKNOWN". The mark of an "UNKNOWN" node is changed in only one place, when the node is the successor of some other node. Thus, if a successor of X is not "UNKNOWN", it must be the successor of some node other than X, which is impossible because the graph is a tree. Hence, every successor of X is "UNKNOWN". QED

(19) At this point, notice that the only marking that is really used is the "UNCHECKED" marking. Thus, the mapping can be simplified somewhat by getting rid of references to the other two possibilities (i.e., making them implicit instead of explicit):

```
Initial state:
  MARK[R] = "UNCHECKED"
Each step:
  X := closest node in MARK-1["UNCHECKED"]
  undo MARK[X]
  send X to consumer
  for all successors, Y, of X:
    MARK[Y] := "UNCHECKED"
```

(Note: if MARK[X] is not "UNCHECKED", we will consider it to be undefined.)

(20) The MARK mapping can be inverted (call the inverted mapping MARK'), so we have:

```
Initial state:
  MARK'["UNCHECKED"] = {R}
Each step:
  X := closest node in MARK'["UNCHECKED"]
  remove X from MARK'["UNCHECKED"]
  send X to consumer
  for all successors, Y, of X:
    add Y to MARK'["UNCHECKED"]
```

(21) The MARK' mapping can be represented as a record structure, with one field for each domain element. And since there is only one domain element of concern, "UNCHECKED", there is only one field, so we might as well just use the set itself, and call it Q. Thus, the producer is:

```
Initial state:
  Q = {R}
Termination test:
  is Q empty
Each step:
  X := closest node in Q
  remove X from Q
  send X to consumer
  for all successors, Y, of X:
    add Y to Q
```

(22) If Q is represented as a sequence, ordered by distance from the root, the "closest" node will be the first element:

X := first element of Q

(23) In a sequence, a removal operation normally requires searching for the location in which X is stored, but since X was taken as the first element of Q, we can simply remove the object stored in the first location:

remove first element of Q

(24) In a sequence, an addition operation normally requires searching for a suitable position to insert the new element, but in this case we can show that the back is a suitable position, since Y follows every element in Q according to the ordering constraint.

insert Y at back of Q

With the following lemma, the necessary proof is fairly simple.

LEMMA: for some X in Q,
 $D(R, X) \leq D(R, Z) \leq D(R, X) + 1$ for all Z in Q

This is clearly true initially, since $Q = \{R\}$. Suppose it is true at some point. There are only two changes in Q that can be made during one step:

- (a) X is removed from Q
- (b) Y is added to Q

(a) After X is removed, either Q is empty (in which case adding Y restores the desired property) or there is some new closest node, call it W. We know:

$D(R, X) \leq D(R, W) \leq D(R, Z)$ for all Z in Q

so substitution (into the inequality that held before X was removed) gives us:

$D(R, W) \leq D(R, Z) \leq D(R, W) + 1$ for all Z in Q

(b) After Y is added, since $D(R, Y) = D(R, X) + 1$,

$D(R, Z) \leq D(R, X) + 1 = D(R, Y) \leq D(R, W) + 1$ for all Z in Q

Thus, by induction we have:

for some X in Q,
 $D(R, X) \leq D(R, Z) \leq D(R, X) + 1$ for all Z in Q

QED

We are now ready to prove the condition necessary for adding Y, a successor of X, at the back of Q.

PROOF: Y follows every element of Q

The ordering constraint on Q is:

if $D(R,X) < D(R,Z)$ then X precedes Z in Q

So what we wish to show is:

if $D(R,X) \leq D(R,Z)$ for all Z in Q,
and Y is a successor of X,
then $D(R,Z) \leq D(R,Y)$ for all Z in Q.

Since the successors of X are all the same distance from X (and hence from the root), their relative order is unimportant; thus, showing the above is sufficient to enable us to insert them one at a time at the back.

The lemma indicates that there is some X in Q such that everything else in Q is at least as far from the root as X, but not further away than 1 arc. Therefore, this property must hold for the closest node in Q. Since $D(R,Y) = D(R,X) + 1$, if Y is a successor of X, we know that $D(R,Z) \leq D(R,Y)$ for all Z in Q. Thus, Y follows every element of Q. QED

(25) The producer is finally finished:

```
Initialization:
  Q := <R>
Termination test:
  is Q empty
Body:
  X := first element of Q
  remove first element from Q
  send X to consumer
  for all successors, Y, of X:
    insert Y at the back of Q
```

(26) One simple way to combine a producer and a consumer is to use a WHILE loop:

```
S := <>
Q := <R>
while Q is not empty do
  X := first element of Q
  remove first element from Q
  insert X at back of S
  for all Y in CHILDREN(X) do
    insert Y at back of Q
```

From this point on, rules about simple symbolic programming (such as those in PECOS) could produce the final implementation. The interesting aspects involve representing the sequences S and Q (each is probably best represented as a linked list with a special pointer to the last cell), and the "for all" construct (which depends on the representation of the set returned by CHILDREN(X)).

3. TOWARD AN IMPLEMENTATION

As noted earlier, the hypothetical synthesis presented above is beyond the capabilities of any current automatic programming system. Nonetheless, it seems a reasonable goal for the near future. We are currently attempting to codify programming knowledge about elementary graph algorithms, and breadth-first enumeration is one of our target programs (along with such graph problems as minimum-cost spanning tree, reachability, and connectedness) [1]. The preliminary design of our system (called PKX, for Programming Knowledge eXperiment) reflects the roles of knowledge and deduction discussed here. In designing and implementing PKX, several issues seem particularly important.

One of these is simply: What knowledge about graph algorithms should be represented? Much programming knowledge is currently available only informally (or even subconsciously) in the heads of programmers, and it must be explicated to a rather detailed level if it is to be usable by a machine. The hypothetical synthesis above suggests some of the necessary knowledge. For example, step (9), which introduced the MARK mapping, is based on knowing a particular scheme for saving the state of an enumeration of the nodes in a graph. But this knowledge must still be reduced to some kind of rule form, and organized into a usable data base. In trying to use this knowledge, two other issues arise. PECOS's refinement paradigm was fairly successful and convenient for exploration, primarily because of the rarity of dead-ends. In PKX, especially when dealing with non-algorithmic specifications, this is unlikely to be the case, so we expect more search to be involved. (And the cost of search is higher because of the cost of the deductive system proving some rule applicability conditions.) Another kind of search involves selecting the "best" implementation from among the successful paths. In order to control both kinds of search, we plan to make extensive use of plausibility and preference rules, such as those of LIBRA [5].

As far as the deductive system is concerned, if we take the five proofs in the above synthesis as representative, the complexities of the deductions seem to be generally comparable to those of state-of-the-art theorem provers. The one exception is that of step (24), which involved a lemma in addition to the main proof. In order to experiment with axiomatizations of the relevant concepts, we have built an experimental resolution theorem prover, and were able to construct proofs for steps (5) and (16). The proofs for steps (13) and (18) both involve changes made to a data structure, a topic that will require more effort to axiomatize adequately. Of course, various properties of graphs and trees, as well as operations on them, will also have to be axiomatized. Perhaps the most important conclusion from our experiment with a resolution theorem prover was that it was too undirected for our purposes, so we plan to design and build a more special-purpose deductive system for use with PKX.

4. CONCLUSIONS

In order for future automatic programming systems to be truly useful when applied to real-world tasks, they will necessarily have to deal with complex programs at a rather detailed level. It seems to me that this will only be possible if these systems have effective access to knowledge about what we today consider to be the task of programming. While some of this knowledge certainly involves general strategies, such as the conditional and recursion introduction rules of DEDALUS, much of the knowledge also involves rather specific detailed facts about programming in particular domains, such as PECOS's rules about symbolic programming. (This may, in fact, merely be another restatement of the trade-off between generality and power.) The first attempts to build such programming knowledge into an automatic programming system involved applying the knowledge to algorithmic specifications, in which the user described the program abstractly and the system produced a concrete implementation. The hypothetical synthesis of a breadth-first enumeration program suggests that specific detailed knowledge about programming can also be of significant value for non-algorithmically specified programs. It also illustrates an important role to be played by deduction in such knowledge-based automatic programming systems: as a mechanism for answering particular queries about the program being constructed, as part of the process of testing the applicability of particular rules. This auxiliary role for deduction seems an important one for future development.

ACKNOWLEDGEMENTS

Much of this work resulted from discussions during seminars in Knowledge-based Automatic Programming held at Yale in Fall 1977 and Spring 1979. Mark Zbikowski built the resolution theorem prover. Chuck Rich, Howie Shrobe, Dick Waters, and Richard Waldinger provided helpful comments on an earlier draft of the paper.

REFERENCES

- [1] Barstow, D.R. "Codification of programming knowledge: graph algorithms," Yale University, Department of Computer Science, TR 149, December 1978.
- [2] Barstow, D.R. "An experiment in knowledge-based automatic programming," Artificial Intelligence, (to appear) 1979.
- [3] Barstow, D.R. Knowledge-based Program Construction, Elsevier North Holland, 1979.
- [4] Green, C.C. "The application of theorem proving to question-answering systems," Stanford University, Computer Science Department, AIM-96, August 1969.
- [5] Kant, E. "A knowledge-based approach to using efficiency estimation in program synthesis," Sixth International Conference on Artificial Intelligence, (to appear) 1979.
- [6] Manna, Z. and Waldinger, R. "Knowledge and reasoning in program synthesis," Artificial Intelligence, 6, 1975, 175-208.
- [7] Manna, Z. and Waldinger, R. "Synthesis: dreams => programs," SRI International Technical Note 156, 1977.
- [8] Rich, C. and Shrobe, H.E. "Initial report on a LISP programmer's apprentice," IEEE Transactions on Software Engineering, 4, November 1978, 456-467.
- [9] Simon, H.A. "Experiments with a heuristic compiler," Journal of the ACM, 10, April 1963, 493-506.
- [10] Waldinger, R. and Lee, R.C.T. "A step toward automatic program writing," International Joint Conference on Artificial Intelligence, Washington, D.C., 1969, 241-252.