THE SYNTHESIS OF PROGRAMS BY ANALOGY[1]

Robert Moll
Department of Computer and Information Science
University of Massachusetts
and
John Wade Ulrich
Department of Computing and Information Sciences
University of New Mexico
Albuquerque, New Mexico

COINS Technical Report 79-03

1 This work is supported by NSF Grant NSF-MCS78-23505 and NSF Grant NSF-MCS78-04746. Some of this research was done during January, 1979 while the authors were visiting scientists at SRI International, Menlo Park, California.

THE SYNTHESIS OF PROGRAMS BY ANALOGY[1]

Robert Moll
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts

and

John Wade Ulrich
Department of Computing and Information Science
University of New Mexico
Albuquerque, New Mexico

ABSTRACT

We describe a method for constructing recursive programs. Program specifications are expressed in a LISP-like language, augmented by the quantifiers 'for all', 'there exists', and two other non-constructive operators, 'find' and 'findlist'. Programs are created by applying a sequence of transformations to specifications. The transformational sequence eliminates the non-constructive operators and quantifiers while preserving the meaning of the specifications. The result is a legal program in our LISP-like base language.

A transformational sequence is called a synthesis plan. While the transformations in a plan may reference expressions from the vocabulary of the specifications, no transformation explicitly introduces knowledge which is specific to a particular problem. In this way a single plan may be applicable to a number of programming problems which share certain common syntactic features. An analogical mapping between vocabularies of two problem specifications will transform a successful synthesis plan for one problem into a successful plan for the second problem.

[1] This work is supported by National Science Foundation grants MCS78-23505 and MCS78-04746. Some of this research was done during January, 1979, while the authors were visiting scientists at SRI International, Menlo Park, CA.

INTRODUCTION

Automatic program synthesis is the mechanical development of programs from non-constructive specifications. In this paper we describe a program synthesis method that develops programs by applying a sequence of transformations to their specifications. Specifications are formulated in a LISP-like language augmented by the quantifiers 'all', 'exists', and operators 'find', and 'findlist'. For example, the predicate that determines whether or not a list contains only atoms may be specified with the expression:

```
islat(L) ← all z (member (z,L) implies atom(z)) where islist(L).
```

Our synthesis technique is a quantifier elimination method not unlike the quantifier elimination methods of mathematical logic. In our framework a quantified expression indicates a non-constructive search. In the case of the find operator this search is mentioned explicitly:

```
find z (member (z,L) and atom(z)).
```

From this point of view, program synthesis may be regarded as the replacement of an expression involving quantifiers and find operations with a program segment. The transformations of our system accomplish this replacement by refining the case structure of the specifications or by logically simplifying specification subexpressions. If the transformations are chosen properly and are applied in the correct order, then all of the non-constructive expressions in the specification will be eliminated and the result will be a legal program.

The transformations in our method are quite similar to transformations employed in the work of Manna and Waldinger [Manna 1, Manna 2] and Darlington and Burstall [Burstall, Darlington]. We also employ a simplifier similar in spirit to the Boyer-Moore theorem prover [Boyer] and the Oppen-Nelson simplifier [Nelson, Oppen].

The method described in this paper has been implemented in LISP. Our program can extract from a particular sequence of transformations a more general object which we call a synthesis plan. With appropriate changes in vocabulary, the program can successfully apply a given plan to a number of other programming problems.

The paper is divided into 3 sections. In Section I we give an informal treatment of two programming problems solved by our synthesis method. In Section II we give a general discussion of our method and the transformations it employs. In Section III we consider our synthesis method as a formal language, and we discuss the role of analogical reasoning in the context of this language.

I. SAMPLE PROGRAMMING PROBLEMS

We illustrate our synthesis method with two examples. The first of these is a predicate islat(L). Islat(L) is true if and only if the list L is a list of atoms. Islat(L) is formulated in our specification language by the expression:

islat(L) ← all z [member(z,L) implies atom(z)], where islist(L).

The word 'where' is used to identify the precondition of the program.

Step 1: Case L = ( ) [( ) denotes the empty list]

The body of the program specification is replaced by

```
if  L = ( ) then (all z [member(z,L) implies atom(z)])
  else (all z [member (z,L) implies atom(z)]).
```

Explanation:

The synthesis rule 'case B' causes the replacement of an expression A with an expression 'if B then A else A'.

Step 2: Unfold member(z,L) and Simplify.

The body of the specification becomes:

```
if L = ( ) then true else all z [member (z,L) implies atom(z)]
```

Explanation:

The unfolding rule replaces member(z,L) with the body of the membership function (car(L) and cdr(L) identify the head and tail of the list L).

```
  if L = ( ) then false else if car(L) = z then true
     else member(z, cdr(L)), where islist(L).
```

Under the case assumption L = ( ), member(z,L) simplifies to false. The simplifier further simplifies all z(false implies atom(z)) to true.

Step 3: Case atom(car(L))

```
if L = ( ) then true else
   if atom(car(L)) then (all z [member(z,L) implies atom(z)])
     else (all z [member(z,L) implies atom(z)]).
```

Step 4: Unfold member(z,L) and Simplify

```
if L = ( ) then true else
   if atom (car(L)) then (all z [member(z, cdr(L)) implies atom (z)])
     else.....
```

Explanation:

Membership is unfolded once again, and simplification is performed under the assumption atom(car(L)).

Step 5: Fold

```
if L = ( ) then true else
   if atom(car(L)) then islat(cdr(L)) else
      (all z [member(z,L) implies atom(L)]).
```

Explanation:

The expression (all z[member(z, cdr(L)) implies atom(z)]) matches the top level specification of the program, with cdr(L) as a replacement for L. We replace the quantified expression with the recursive call islat(cdr(L)).

Step 6: Unfold member(z,L) and Simplify

```
if L = ( ) then true else
   if atom(car(L)) then islat(cdr(L)) else false
```

Explanation:

The final occurrence of the membership predicate is unfolded, and the expression is simplified to false under the assumption not(atom(car(L))). The result is the desired program.

The second example is the synthesis of a function which has as its value a list of even numbers. The numbers on the list are to belong to a second list x. We have the following definition:

```
evenof(x) ← findlist (y)[even(y) and member(y,x)]
        where listofnumbers (x)
```

Explanation:

The operation findlist (y) used in the above definition specifies the list to be found. y identifies a typical element of the list.

Step 1: case x = ( )

The body of the program specification is replaced with

```
if x = ( ) then findlist (y) [even(y) and member(y, ( ))] else
   findlist (y) [even(y) and member(y,x)] where not x = ()
      and listofnumbers (x)
```

Step 2: Unfold member(y,x) and Simplify

The body of the program specification becomes

```
if x = ( ) then ( ) else findlist (y) [even(y) and member(y,x)]
    where not x = ( ) and listofnumbers (cdr(x))
```

Step 3: Test car(x)

The transformed specification becomes

```
if x = ( ) then ( ) else
    if even(car(x)) and member(car(x), x)
        then cons(car(x), findlist (y) [even(y) and member(y,x)])  else
           findlist (y) even(y) and member(y,x) where
               not x = ( ) and not y = car(x) and listofnumbers(cdr(x))
```

Explanation:

The test car(x) transformation causes the code to be inserted which tests the car(x) and inserts it into the list if car(x) is an even number.

Step 4: Simplify

```
if x = ( ) then ( ) else
  if even(car(x)) then [cons(car(x), findlist (y) even(y) and member(y,cdr(x))]
     where not y = ( ) and not y = car(x)
         else findlist (y) even(y) and member(y,cdr(x))
              where not x = ( ) and not y = car(x)
```

Explanation:

The simplification deleted 'member(car(x), x)' because 'member(car(x), x)' is true, and 'true and even(car(x))' simplifies to 'even(car(x))'. 'member(y, x)' becomes 'member(y, cdr(x))' because of the assumption 'not y = car(x)'.

Step 5: Fold

```
if x = ( ) then ( ) else if even(car(x)) then
   cons(car(x), even(cdr(x))) else even(cdr(x))
```

Explanation:

The recursive calls were allowed because the preconditions of the program were met and the post conditions were an instance of the output expression in the program specification.

PART II. THE TRANSFORMATIONAL METHOD

General Strategy

The general goal of our transformational method is quantifier elimination and the removal of finds and findlists. There are two ways that quantifiers and finds are removed from the specifications. The first is by applying one of the basic trivial-quantifier rules. These rules drop quantifiers when the quantified variable fails to occur in the expression under the quantifier, or when a trivial expression (eg., all x[x = x]) occurs. The second way that quantifiers and finds are removed is by the introduction of calls to subprograms. An important special case occurs when the specifications have been reduced to an instance of the original specifications. In this case a recursive call is used to replace the quantified expression. The other transformations used by the system try to bring about one of these two situations. We will now discuss each of the transformations.

Simplification

The 'simplify' transformation performs a logical simplification of the current expression. This can be accomplished in one of two ways. The first is the usual propositional simplification (i.e. 'A and A' may be replaced with 'A' etc.). The second involves the distribution of quantifiers over subexpressions: 'all x P(x) and Q(x)' becomes 'all x P(x) and all x Q(x)'. The rules that govern the distribution of quantifiers over subexpressions follow closely those of standard logic.

Case Introduction

The 'case test' transformation causes a proposition 'test' to be inserted into the partial solution by replacing an expression exp with "if test then exp else exp'. The two occurrences of exp are then subjected to further simplification under the hypothesis that test is true and test is false respectively.

Special Instances of Quantified Variables

The transformation 'test term' causes particular instances of quantified variables to be introduced into the partial solution. Suppose that exp has the form 'all x P(x)'. Then test would cause exp to be replaced with 'if P(car(y)) then for all x P(x) where not x = (car(y)) else false'. The precondition 'where not x = (car(y))' serves to modify the quantifier.

Folding and Unfolding

Folding and unfolding are complimentary mechanisms. The instruction 'unfold' causes an application of a previously synthesized function to be replaced with the body of its program after appropriate substitutions are made for its formal parameters. Folding supplies a procedure call to replace a quantified expression which satisfies the post-conditions of a procedure's definition. To see if folding is possible, the quantified expression is matched against a program's definition. If, under appropriate substitutions, the post-conditions of the expression are implied by the post-condition of the definition and the pre-conditions of the expression imply the pre-conditions of the definition, then the expression is replaced by a procedure call.

In addition to seeing that the conditions of the folded program satisfy the folded expression, it is necessary to insure that an infinite recursion will not be introduced by folding. This is accomplished by verifying that the actual parameters come before the formal parameters in a standard well founded ordering based on size of the LISP data structures.

Generalization

The word generalization usually refers to the process by which a problem becomes thought of as a particular instance of a more general problem. In programming this is accomplished by introducing a new function application into the program. This new function is defined in such a way that the application within the program is guaranteed to have the desired value. As an example suppose that exp = 'all z (member(z, cdr(x)) implies car(x) < z)'. Then exp may be replaced with 'F(car(x), cdr(x))' where F is defined as follows: F(y1, y2) $\leftarrow$ all z (member(z, y2) implies y1 < z).

Conjunctive Goals

A synthesis can become blocked when the expression to be transformed has the form 'find z P(z, x) and Q(z, x)', and the conditions P(z, x) and Q(z, x) fail to match any of the definitions of previously synthesized programs. As a last resort we attempt to define a pair of new functions which will solve the 'find z P(z, x) and Q(z, x)' problem. Let us call these two functions F1 and F2. The definitions of these two functions are based on Djikstra's technique for solving conjunctive goals. The idea is basically this: if you want to satisfy two conditions P and Q simultaneously, first satisfy P. Then keep P invariant and try to establish Q. The technique, which is basically an iterative one, may be adapted as follows: first define the program F1 so that its output satisfies Q(z, x). Then pass the output from F1 to a second program F2 which tries to establish both P(z, x) and Q(z, x). It may seem that no progress has been made because the second program must still find a value which satisfies P(z, x) and Q(z, x). The advantage is that the pre-conditions of F2 are stronger than those of the original problem in that a particular value is known which satisfies Q.

We define F1 and F2 as follows:

```
    F1(x) ← find z Q(z, x)
    F2(x, y) ← if P(y, x) then y else
          (find z P(z, x) and Q(z, x) where Q(y, x) and not P(y, x)).
```

Our test 'if P(y, x)' in F2 comes from the common sense rule that says that if y satisfies Q, and Q and P must both be true, then first find out if y also satisfies P.

III. ANALOGICAL REASONING AND SYNTHESIS PLANS

In our exposition of the synthesis of islat(L) and evenof(L), we showed how the application of a small set of transformations to program specifications can accomplish program synthesis. We now consider how such a sequence of transformations might be modified to apply to a second programming problem. Certain transformations, such as case introduction, refer to expressions in the specification vocabulary of the original problem. These expressions must be replaced with analogous expressions in the vocabulary of the second problem. We establish this correspondence on the basis of a syntactic comparison between the specifications of the two problems. From this point of view, the synthesis sequence of the original problem may be thought of as a plan for the synthesis of the second problem, as well as for other problems whose specifications are syntactically similar to the original specification.

Suppose we consider the transformational sequence that derived the islat program. Recall that islat has the specification

```
    islat(L) ← all z [member(x, L) implies atom(x)] where islist(L))
```

We can summarize the transformational sequence that derived the islat program using the following notation, which imposes a program-like character on our transformation languages.

```
    case L = ( )
         true:  <unfold member(z, L), simplify>
         false:  <unfold member(z, L);
                      case atom(car(L))
                         true:  <simplify, fold>
                         false:  <simplify>>
```

Now suppose we consider a problem which is quite similar to the islat synthesis problem: the synthesis from specifications of the predicate issubset(L1, L2). Issubset(L1, L2) has the specification

issubset(L1, L2) $\leftarrow$ (all z (member(z, L1) implies member(z, L2))
where islist(L1) and islist(L2).

This specification is syntactically analogous to the islat specification, and we can indicate the relevant analogical correspondences as follows:

L	:	L1
member(z, L)	:	member(z, L1)
atom(z)	:	member(z, L2)

Our next task is to attempt the synthesis of issubset using the islat synthesis plan and the analogical correspondences mentioned above. If we systematically replace the islat expressions with the corresponding issubset expressions we are left with the following plan for issubset:

```
         case L1 = ( )
             true:  <unfold member(z, L1), simplify>
            false:  <unfold member(z, L1);
                          case member(car(L1), L2)
                             true:  <simplify, fold>
                            false:  <simplify>>
```

When this plan is applied to the issubset specifications, the following program is produced:

```
                  issubset(L1, L2):
                       if L1 = ( ) then true else
                           if member(car(L1),L2) then issubset(cdr(L1), L2) else
                              false.
```

As a second example of this principle consider the problem of determining if every member of a list L1 is less than some member of a second list L2. This problem can be specified as follows:

```
          maxlists(L1, L2) ← all z[member(z, L1) implies
               exists w [member(w, L2) and  z<w)]],
                 where islist(L1) and islist(L2).
```

Once again we indicate a family of analogical correspondences between the target specification and the islat specification:

L	:	L1
member(z, L)	:	member(z, L1)
atom(z)	:	exists w (member(w, L2) and z<w)

We may introduce the abstraction:

```
          bounded(w, L2) ← exist w (member(x, L2)) and z<w where islist(L2)).
```

If we systematically substitute the maxlists expressions for the corresponding islat expressions in the islat synthesis plan we arrive at a successful plan for the synthesis of maxlists:

```
          case L = ( )
             true:  <unfold member(z, L1); simplify>
            false:  <unfold member(z, L1);
                         case bounded(car(L1), L2)
                              true: <simplify, fold>
                             false: <simplify>>
```

The code that is generated is:

```
maxlists(L1, L2);
     if L1 = ( ) then false else
        if bounded(car(L1), L2) then maxlists(cdr(L1), L2)
             else false.
```

In this case synthesis is only partially complete. A subproblem, the synthesis of bounded(z, L1), must be solved in order to complete the synthesis.

We point out that the program synthesized is not the most efficient algorithm for computing maxlists. A more efficient algorithm would compute the max of L1 first, and then check to see if some element of L2 exceeded this number.

CONCLUSIONS

We have formulated a framework for studying analogical reasoning as a tool for automatic program synthesis. For this purpose we have developed an abstract representation of synthesis plans. When a plan is applied to an appropriate program specification, a program is produced which meets those specifications. A single plan is applicable to a variety of programming problems whose specification have common syntactic features.

It might be expected that analogical mappings based on syntactic comparisons would not be particularly useful for the synthesis of programs, since the semantics of syntactically similar specifications may be quite different. This has not turned out not to be the case. The transformations described in this paper have been implemented and have worked successfully on a set of example problems, including the ones used as illustrations in this paper.

The ability to acquire knowledge from examples has obvious advantages over other methods of knowledge acquisition. Our future work will involve the construction of a data base of plans and associated specifications. The system will be trained on a set of example programs, after which it will be expected to solve new problems based on plans derived from the training set.

BIBLIOGRAPHY

[Dershowitz] Dershowitz, N., and Manna, Z., The Evolution of Programs: A System for Automatic Program Modification, IEEE Transactions on Software Engineering, Vol. SE-3, No.4, 1977.

[Manna 1] Manna, Z., and Waldinger, R.J., "Knowledge and Reasoning in Program Synthesis', Artificial Intelligence, Vol. 6, No. 2, 1975, pp. 175-208.

[Manna 2] Manna, Z., and Waldinger, R.J., "Synthesis: Dreams → Programs Technical Note 156, SRI International, Menlo Park, California, 1977.

[Nelson] Nelson, G., and Oppen, D. "Simplification by Cooperating Decision Procedures", Fifth ACM Symposium on Principles of Programming Languages, 1978.

[Boyer] Boyer, R.S., and Moore, J.S., "A Lemma Driven Automatic Theorem Prover for Recursive Function Theory", Proceedings of the Fifth IJCAI, Cambridge, Massachusetts, 1977, pp. 511-519.

[Burstall] Burstall, R.M., and Darlington, J., "A Transformation System for Developing Recursive Porgrams", JACM, Vol. 24, No.1, 1977, pp 46-67.

[Darlington] Darlington, J., "A Synthesis of Several Sorting Algorithms", Research Report 23, Department of Artificial Intelligence, University of Edinburgh, Scotland, July, 1976.

[Ulrich] Ulrich, J.W., and Moll, R., "Program Synthesis by Analogy", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, Rochester, New York, 1977, pp 22-28.

[Oppen] Oppen, D., "Reasoning about Recursively Defined Data Structures", Stanford Artificial Intelligence Laboratory, Memo AIM-314, July, 1978.