

THE SYNTHESIS OF PROGRAMS BY ANALOGY

Robert Moll
Computer and Information Science Department
University of Massachusetts and
Hampshire College
Amherst, MA 01003

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

Analogical reasoning is the method by which solutions to old problems are adapted to solve new problems. In this paper we illustrate how analogy can be used in a non-trivial problem solving domain. Though we restrict ourselves here to the domain of computer programming, we argue that the principal idea is applicable to a broad class of problem solving situations.

1. PLANS AND ANALOGIES

A plan is a description of how a problem may be solved. We wish to adapt old plans to new problems. To do this we first identify certain details of a plan that are particular to one problem, and replace them with details from a second problem. This plan modification process is the essential part of our formulation of analogical reasoning. Reasoning by analogy is successful if application of the modified plan solves the new problem.

In order to automate analogical reasoning in the context of a transformational program synthesis system, plans must be described so that programming details that apply to particular problems may be easily identified and modified. We accomplish this by first specifying a small set of elementary transformations. Each elementary transformation, when appropriately instantiated, is applicable to a variety of programming situations. Then we introduce a means for combining elementary transformations into plans. The following is an example of a synthesis plan:

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

The underlined expressions are formed from the predicates and terms of the specifications for the program islat(L), which returns true if the list L contains only atoms:

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

The words simplify, fold, case, test and unfold refer to elementary transformations. A plan is executed by applying the elementary transformations to selected portions of the program specification. If the final result is a well-formed program then the plan is successful. In the islat synthesis example the elementary transformations are applied as follows:

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:
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 (all z)[member(z,L) implies atom(z)].
```

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(z)].
```

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.

2. SYNTHESIS BY ANALOGY

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,L) 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)  $\leftarrow$  (all z)[member(z,L) implies
  (exists w)[member(w,L2) and z < w]],
  where islist(L1) and islist(L2).
```

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

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

We may introduce the abstraction:

```
bounded(w,L2)  $\leftarrow$  (exists 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.

3. CONCLUSIONS

We have developed a framework for studying analogical reasoning as a tool for automatic program synthesis, and we have implemented an interactive synthesis system. Our system has synthesized a variety of programming problems including those in the text. In addition to universal and existential quantifiers our specification language allows two other non-constructive operators, 'find' and 'findlistof'.

In order to give a mechanical formulation of analogical reasoning we have found it essential to distinguish between knowledge about particular problems and knowledge about problem solving. The first kind of knowledge is represented in the form of definitions and specifications. The second kind of knowledge is represented in the form of plans. The bridge between these two types of knowledge is the vocabulary of the problem being solved. If the design of a transformation-based system preserves the distinction between types of knowledge, then analogical reasoning as a problem solving device becomes possible. That is, a particular problem solving strategy, represented as a transformational

sequence or plan, will have applicability to a variety of intuitively similar problems. The distinction between domain knowledge and problem solving strategies can be enforced by requiring that transformation rules know only about definitions and how they may be manipulated. This is contrary to the frequent practice of embedding domain knowledge in the rules of a problem solving system.

4. BIBLIOGRAPHY

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

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

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

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

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

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

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.