PROGRAMMING BY ANALOGY

Nachum Dershowitz*
Department of Computer Science
University of Illinois
Urbana, IL 61801

ABSTRACT

Analogy is one tool that automatic programming systems can use to learn from experience, just as programmers do. We illustrate how analogies between program specifications can be used to debug incorrect programs, modify existing programs to perform different tasks, derive abstract schemata from given sets of cognate programs, and instantiate schemata to solve new problems.

An analogy between the specification of a given program and that of a new problem is used as the basis for modifying the given program to meet the new specification. Debugging is a special case of modification: if a program computes wrong results, it must be modified to achieve the intended results. For program abstraction, an analogy is sought between the specifications of the given programs; it may then be used to transform an existing program into an abstract schema that embodies the shared technique. By comparing the specification of the derived schema with a given concrete specification, and formulating an analogy between them, an instantiation of the schema may be found that yields the desired concrete program.

Key terms: learning, analogy, automatic programming, program modification, debugging, abstraction, instantiation, program transformations, program schemata.

> ***Analogy pervades all our thinking, our everyday speech and our trivial conclusions as well as artistic ways of expression and the highest scientific achievements.***
>
> ***—George Polya***

1. INTRODUCTION

Programming begins with a specification of what the envisioned program ought to do. It is the programmer's job is to develop an executable program satisfying that specification. Yet, only a small fraction of a programmer's time is typically devoted to the creation of original programs *ex nihilo*. Rather, most of his effort is normally directed at debugging incorrect programs, adapting known techniques to specific problems at hand, modifying existing programs to meet amended specifications, extending old programs for expanded capabilities, and abstracting ideas of general applicability into "subroutines."

*This research supported in part by the National Science Foundation under Grant MCS-79-04897.

The goal of research in "automatic programming" is to formalize methods and strategies used by programmers so that they may be incorporated in automatic, and interactive, programming environments. In our view, program development systems should incorporate formal tools for transforming and manipulating programs. In this paper, we illustrate how analogies might be used by such a system for that purpose.

The importance of analogical reasoning has been stressed by many, from Descartes to Polya. The use of analogy in automated problem solving was proposed in [Kling71]. Other works employing analogy as an implement in problem solving include [Brown76], [McDermott79], and [Winston80]. The use of analogies to guide the modification of programs was proposed in [MannaWaldinger75] and pursued in [Dershowitz-Manna77] and [UlrichMoll77].

Programmers improve with experience by assimilating programming techniques that are encountered, and judiciously applying the ideas learned to new problems. One way in which knowledge can be applied is by modifying a known program to achieve some new goal. For example, a program that uses the "binary-search" technique to compute square-roots might be transformed into one that divides two numbers. We show how to modify programs by first finding an analogy between the specification of the existing program and that of the program we wish to construct. This analogy is then used as a basis for transforming the existing program to meet the new specification. Program debugging is a special case of modification: if a program computes wrong results, it must be modified to achieve the intended results.

All our programs are assumed to be annotated with an *output specification* (stating the desired relationship between the input and output variables upon termination of the program), an *input specification* (defining the set of legal inputs on which the program is intended to operate), and *invariant assertions* (relations that are known to always hold at specific points in the program for the current values of variables) demonstrating its correctness. The invariant assertions play an important role in deriving analogies.

The idea that programs should be constructed by a series of transformations has been widely promoted. Modification differs from such transformations in that correctness with respect to the original specification is *not* preserved. What we want is for the resultant program to be correct with respect to the *transformed* specification. Correctness-preserving transformations and specification-changing modifications are thus complementary. A scenario of computer-aided programming and debugging appeared in [Floyd71]. The HACKER system [Sussman75] constructed programs by trying out alternatives and attempting to debug them when necessary; other knowledge-based or plan-based

debugging systems have been constructed, as well. [KatzManna75] and [Sagiv76] describe debugging techniques based—like our method—on invariant assertions.

Program modification is not the only manner in which a programmer utilizes previously acquired knowledge. After coming up with several modifications of his first "wheel," he is likely to formulate for himself (and perhaps for others) an abstract notion of the underlying principle and reuse it in new, but related, applications. Program "schemata" are a convenient form for remembering such knowledge. A schema may embody basic programming techniques and strategies (e.g. the "generate-and-test" paradigm or the "binary-search" technique) and contains abstract, uninstantiated symbols, in terms of which its specification is stated.

The abstraction of a set of concrete programs to obtain a program schema and the instantiation of abstract schemata to solve concrete problems may be viewed from the perspective of modification methods. This perspective provides a methodology for applying old knowledge to new problems. Beginning with a set of programs sharing some basic strategy and their correctness proofs, a program schema that represents their analogous elements is derived. Preconditions for the schema's applicability are derived from the correctness proofs. The resultant schema's abstract specification may be compared with a given concrete specification to suggest an instantiation that yields a concrete program when applied to the schema. If the instantiation satisfies the preconditions, the correctness of the new program is guaranteed.

To date there has been a limited amount of research on program abstraction. The STRIPS system [FikesHartNilsson72] generalized the loop-free robot plans that it generated; HACKER "subroutinized" and generalized the "blocks-world" plans it synthesized, executing the plan to determine which program constants could be abstracted. [DershowitzManna75] suggested using the proof of correctness of a program to guide the abstraction process; that idea was pursued further in [Dershowitz81]. [Gerhart75] and others have advocated and illustrated the use of schemata as a powerful programming tool. A collection of such schemata, along with a catalog of correctness-preserving program transformations, could serve as part of an interactive program-development system.

In the following sections we trace the life-cycle of an example program. The example illustrates some of the kinds of transformations programs undergo and how analogy can be used as a guide. We begin with an imperfect program to compute the quotient of two real numbers. After the program is *debugged,* it is *modified* to approximate the cube-root of a real number. Underlying both the division and cube-root programs is the binary-search technique; by *abstracting* these two programs, a binary-search schema is obtained. This schema is then *instantiated* to obtain a third program, one to compute the square-root of an integer.

2. DEBUGGING

Consider the problem of computing the quotient q of two nonnegative real numbers c and d within a specified (positive) tolerance e. These requirements are conveniently expressed in the form of the following skeleton program:

```
P1: begin comment real-division specification
    assert 0≤c<d, e>0
    achieve |c/d−q|<e varying q
    end
```

The **achieve** statement,

achieve $|c/d-q|<e$ **varying** q,

contains the *output specification* which gives the relation between the variables q, c, d, and e that we wish to be attained at the end of program execution: the (absolute value of the) difference between the exact quotient c/d and the result q should be less than e. The clause **varying** q indicates that of the variables in the specification, only q may be set by the program; the other variables, c, d, and e, contain input values that remain fixed. The *input specification* defines the set of inputs on which the program is intended to operate. Assuming that we wish our program to handle the case when the quotient is in the range 0 to 1, that is when the numerator c is smaller than the denominator d, the appropriate input specification is contained in the **assert** statement,

assert $0\le c\le d,\ e>0$,

attached to the beginning of the program. For the problem at hand, we assume that no general real-division operator / is available, though division by powers of two ("shifts") is permissible.

Now let us imagine that a programmer went ahead and constructed the following program:

```
T2: begin comment suggested real-division program
  B2: assert 0≤c<d, e>0
  purpose |c/d−q|<e
    purpose q≤c/d<q+s, s≤e
      (q,s) := (0,1)
      loop L2: suggest q≤c/d<q+s
           until s≤e
           purpose q≤c/d<q+s, 0<s<sL2
             if d·(q+s)≤c then q := q+s fi
             s := s/2
           repeat
    suggest q≤c/d<q+s, s≤e
  E2: suggest |c/d−q|<e
  end
```

The **purpose** statement,

purpose $|c/d-q|<e$,

is a comment describing the intent of the code following it. The statement

suggest $|c/d-q|<e$

contains the programmer's contention that the preceding code actually achieves the desired relation, i.e. the relation $|c/d-q|<e$ holds for the value of q when control reaches the end of the program. The comment

purpose $q\le c/d<q+s,\ s\le e$

indicates that the programmer's intention is to achieve the desired relation $|c/d-q|<e$ by achieving the subgoals $q\le c/d\le q+s$ and $s\le e$. Achieving these relations is sufficient for $|c/d-q|<e$ to hold. For this purpose the programmer

constructed an iterative loop intended to keep the first relation invariantly true while making progress towards the second. The suggested invariant is contained in the statement

suggest $q \le c/d < q+s$

at label L_2. The goal of the loop body is

purpose $q \le c/d < q+s,\ 0 < s < s_{L_2}$,

where s_{L_2} denotes the value of the variable s when control was last at the label L_2. This means that the value of s is to be less than it just was at the head of the loop. The two loop-body statements are accordingly repeated (zero or more times) until the test $s \le e$ becomes true, at which point the loop will be exited.

We know what the above program was intended for, and we know that it does not always fulfill those intentions. However, before we can debug it, we need to know more about what it actually does. This can be accomplished by examining the code and annotating the program with the discovered invariant relations (see [DershowitzManna81]). It is not difficult to derive the loop invariant $d \cdot q \le c < d \cdot (q+2 \cdot s)$. This remains true when the loop exit is taken; along with the exit test $s \le e$, it implies that upon termination the output invariant $|c/d - q| < 2 \cdot e$ holds. Note that the desired relation $|c/d - q| < e$ is *not* implied.

Now that we know something about what the program does, we can try to debug it. Our task is to find a correction that changes the actual output invariant

assert $|c/d - q| < 2 \cdot e$

to the desired output invariant

suggest $|c/d - q| < e$.

We go about this by first looking for a way to transform the actual invariant into the desired one; we then try to apply the same transformation to the program, hopefully correcting the error thereby. Accordingly, we would like to find an analogy between the actual output invariant and the desired specification; we write

$$|c/d - q| < 2 \cdot e \iff |c/d - q| < e.$$

The obvious difference between the two expressions is that where the first has $2 \cdot e$, the second has just e. So, we can reduce the analogy to simply

$$2 \cdot e \iff e.$$

We can, therefore, transform the insufficient $|c/d - q| < 2 \cdot e$ into the desired $|c/d - q| < e$ by replacing e with $e/2$, i.e. by applying the transformation $e \Rightarrow e/2$.

So far we know that the transformation $e \Rightarrow e/2$, applied to the output invariant $|c/d - q| < 2 \cdot e$, yields the desired output specification $|c/d - q| < e$. That same transformation is now applied to the whole annotated program. The symbol e appears once in the program text: the exit clause $s \le e$ accordingly becomes $s \le e/2$. The symbol also appears four times in the invariants; for example, the input assertion $e > 0$ transforms into $e/2 > 0$ which is equivalent to $e > 0$. The transformed program is

```
P1: begin comment real-division program
  B1: assert 0≤c<d, e>0
  purpose |c/d-q|<e
    purpose q≤c/d<q+2·s, 2·s≤e
      (q,s) := (0,1)
      loop L1: assert d·q≤c<d·(q+2·s)
            until s≤e/2
            purpose q≤c/d<q+2·s, 0<s<s_L1
              if d·(q+s)≤c then q := q+s fi
              s := s/2
            repeat
    assert q≤c/d<q+2·s, 2·s≤e
  E1: assert |c/d-q|<e
  end
```

(We have also changed the program's **purpose** statements to reflect reality.) It can be shown that a transformation such as $e \Rightarrow e/2$ preserves the relation between the program text and invariants, i.e. the transformed assertions are invariants of the transformed program.

3. MODIFICATION

Consider the following specification:

```
Q3: begin comment cube-root specification
  assert a≥0, e>0
  achieve |a^1/3-r|<e varying r
  end
```

We would like to use the corrected real-division program as a basis for the construction of the specified program for computing cube-roots. (We assume, of course, that the cube-root operator is not primitive.) To this end, we first compare the specifications of the two programs. The output specification of the division program is

assert $|c/d - q| < e$,

while the output specification of the desired program is

achieve $|a^{1/3} - r| < e$ **varying** r.

The obvious analogy between the two is

$$\begin{array}{ccc} q & \iff & r \\ c/d & \iff & a^{1/3}, \end{array}$$

i.e. where the former specification has q, the other has r, and where the former has c/d, the other has $a^{1/3}$. One way to obtain a cube-root program from the division program is via the transformations

$$\begin{array}{ccc} q & \Rightarrow & r \\ u/v & \Rightarrow & u^{1/3} \\ c & \Rightarrow & a, \end{array}$$

where by $u/v \Rightarrow u^{1/3}$ we mean that every occurrence of the (general) division operator / is replaced by the cube-root operator applied to what was the numerator. Transformations that involve specific functions such as division, are not, however, guaranteed to yield a correct program, since the program may be based on some property that holds for division, but not for extracting roots. Such transformations are heuristic in nature; they only suggest a possible analogy between the two programs. Indeed, when applied to the division program P_1 we get a program that computes a/d, not $a^{1/3}$. What must be done in such cases is to review the derivation

of the program, expressed by the programmer in **purpose** statements, and see where the analogy breaks down.

The purpose of the division program was $|c/d-q|<e$ which transformed into $|a^{1/3}-r|<e$ as desired. The programmer achieved $|c/d-q|<e$ by breaking it into the subgoals given in the statement

purpose $q \le c/d < q+2\cdot s,\ 2\cdot s \le e,$

part of which became the exit test for the loop and part became a loop invariant. These subgoals transform into

purpose $r \le a^{1/3} < r+2\cdot s,\ 2\cdot s \le e,$

which indeed imply the transformed goal $|a^{1/3}-r|<e$. The purpose of the loop body of the division program was

purpose $q \le c/d < q+2\cdot s,\ 0<s<s_{L_3}.$

In other words, the loop body reachieves the invariant while making progress towards the exit test by decreasing s. The loop-body subgoal of the transformed program, then, is

purpose $r \le a^{1/3} < r+2\cdot s,\ 0<s<s_{L_3}.$

At this point the division program introduces a conditional with the

purpose $q \le c/d < q+s$

and halves s.

It is here that the analogy breaks down. The division program achieves the above purpose in two cases, by testing if $d\cdot(q+s) \le c$ or not. For example, if $d\cdot(q+s) \le c$ does not hold, then $c/d < q+s$, as desired. On the other hand, the fact that $d\cdot(r+s) \le a$ does not hold in the cube-root program tells nothing about $a^{1/3} < r+s$. We look, therefore, for a transformation that makes $d\cdot(r+s) > a$ imply $a^{1/3} < r+s$, or the equivalent to $a < (r+s)^3$. Matching what we have with what we want tells us that the implication would hold if we could transform $d\cdot(r+s) \Rightarrow (r+s)^3$. Thus, where the division program has the function $u\cdot v$, the cube-root program requires v^3. We complete the analogy by adding the transformation

$$u\cdot v \quad \Rightarrow \quad v^3,$$

which is applied to the conditional test.

There remains one problem: a transformed program can only be expected to satisfy the output specification for those inputs that satisfy the transformed input specification. In our case, we can solve this if we can find an alternative manner by which to initialize the invariant $r \le a^{1/3} < r+2\cdot s$ prior to entering the loop. To achieve the subgoal $r \le a^{1/3}$, we can let $r=0$. Then to achieve $a^{1/3} < r+2\cdot s = 2\cdot s$, we can let $s=(a+1)/2$. (This requires additional knowledge about cube-roots.) The complete cube-root program is:

```
Q3: begin comment real cube-root program
  B3: assert a ≥ 0, e > 0
  (r,s) := (0,(a+1)/2)
  loop L3: assert r ≤ a^(1/3) < r+2·s
       until s ≤ e/2
       if (r+s)^3 ≤ a then r := r+s fi
       s := s/2
       repeat
  E3: assert |a^(1/3)-r| < e
  end
```

4. ABSTRACTION

At this point, we have two programs, P_1 for finding quotients and Q_3 for finding cube-roots. Both programs utilize the binary-search technique. It would be nice if one could extract an abstract version of the two programs that captures the essence of the technique, but is not specific to either problem. The resultant abstract program schema could be used as a model of binary search for the solution of future problems.

For this purpose, consider the complete analogy that we found between the specifications of P_1 and Q_3:

$$\begin{array}{rcl} q & \Longleftrightarrow & r \\ u/v & \Longleftrightarrow & u^{1/3} \\ c & \Longleftrightarrow & a \\ u\cdot v & \Longleftrightarrow & v^3\,. \end{array}$$

Since both u/v and $u^{1/3}$ are functions, we try to generalize them to an abstract function $\gamma(u,v)$. Similarly the generalization of $u\cdot v$ and v^3 is another function $\delta(u,v)$. Both q and r are output variables and are generalized to an abstract output variable z; the input variables c and a are generalized to an abstract input variable x. This gives the following set of transformations for generalizing the division program:

$$\begin{array}{rcl} q & \Rightarrow & z \\ u/v & \Rightarrow & \gamma(u,v) \\ c & \Rightarrow & x \\ u\cdot v & \Rightarrow & \delta(u,v)\,. \end{array}$$

Applying these transformations to the specification

achieve $|c/d-q|<e$ **varying** q

of the division program yields

achieve $|\gamma(x,d)-z|<e$ **varying** z.

This will be the abstract output specification of the schema. Substituting the abstract functions γ and δ into their respective positions in the division program P_1, does not, however, result in a schema that will work for all instantiations of γ and δ. This is because the original program relied upon facts specific to multiplication and division. We must therefore determine under what conditions the abstract schema does achieve its specifications.

To begin with, the transformed initialization assignment does not achieve the desired loop invariant. We therefore replace the loop initialization with the subgoal

achieve $\delta(d,z) \le x < \delta(d,z+2\cdot s)$ **varying** z,s,

leaving—for the time being at least—the specifics of how to initialize the loop invariant unspecified. For the loop-body path to be correct, the truth of the invariant must imply that the invariant will hold next time around; this can easily be shown to be the case for any function δ. For the loop-exit path to be correct, we must have that the loop invariants, plus exit test, imply that the output invariant holds. For this to be the case, it suffices to establish the condition

$$\delta(w,u) \le v \equiv u \le \gamma(v,w).$$

In this manner, we have derived a general program schema for a binary search for the value of $\gamma(x,d)$ within a tolerance e:

S_4: **begin comment** *binary-search schema*
  B_4: **assert** $e>0,\ \delta(w,u)\leq v \equiv u\leq\gamma(v,w)$
  achieve $\delta(d,z)\leq x<\delta(d,z+2\cdot s)$ **varying** z,s
  loop L_4: **assert** $\delta(d,z)\leq x<\delta(d,z+2\cdot s)$
    until $s\leq e/2$
    if $\delta(d,z+s)\leq x$ **then** $z := z+s$ **fi**
    $s := s/2$
    repeat
  E_4: **assert** $|\gamma(x,d)-z|<e$
  end

Of course, for this schema to be executable, the function δ appearing in it must be primitive; otherwise, it should be replaced. Similarly, the unachieved subgoal

achieve $\delta(d,z)\leq x<\delta(d,z+2\cdot s)$ **varying** z,s

must be reduced to primitives.

5. INSTANTIATION

The binary-search schema just derived from the division program may be applied to the computation of the square root of an integer. The goal is to construct a program that finds the integer square-root z of a nonnegative integer a:

R_5: **begin comment** *integer square-root specification*
  assert $a\in N$
  achieve $z=\lfloor\sqrt{a}\rfloor$ **varying** z
  end

where the function $\lfloor u \rfloor$ yields the largest integer not greater than u.

We cannot directly match this goal with the output specification of the schema

assert $|\gamma(x,d)-z|<e$ **varying** z.

However, if we expand the goal $z=\lfloor\sqrt{a}\rfloor$, using the definition of $\lfloor u \rfloor$, we get the equivalent goal

achieve $z\leq\sqrt{a}<z+1,\ z\in Z$ **varying** z

(where Z is the set of all integers), i.e. z should be the largest integer not greater than $\sqrt{a}$. Since we know that the schema achieves the two output invariants

assert $z\leq\gamma(x,d)<z+e$,

we can compare these invariants with the above goal. This suggests the transformations

$$\begin{array}{rcl}\gamma(u,v) & \Rightarrow & \sqrt{u}\\ x & \Rightarrow & a\\ e & \Rightarrow & 1\end{array}$$

to achieve $z\leq\sqrt{a}<z+1$. In addition, we will have to extend the program to ensure that the final value of z is a nonnegative integer.

The precondition for the schema's correctness is

assert $e>0,\ \delta(w,u)\leq v \equiv u\leq\gamma(v,w)$;

instantiating it yields

assert $1>0,\ \delta(w,u)\leq v \equiv u\leq\sqrt{v}$.

This condition may be satisfied by taking $\delta(w,u)$ to be u^2. This completes the analogy, and suggests the additional transformation

$$\delta(w,u) \quad \Rightarrow \quad u^2.$$

Applying the instantiation mapping to the schema, we obtain the partially written program:

R_5: **begin comment** *incomplete integer square-root program*
  B_5: **assert** $a\in N$
  achieve $z^2\leq a<(z+2\cdot s)^2$ **varying** z,s
  loop L_5: **assert** $z^2\leq a<(z+2\cdot s)^2$
    until $s\leq 1/2$
    if $(z+s)^2\leq a$ **then** $z := z+s$ **fi**
    $s := s/2$
    repeat
  assert $|\sqrt{a}-z|<1$
  achieve $z\in Z$ **protecting** $z\leq\sqrt{a}<z+1$ **varying** z
  end

This program still contains two unachieved subgoals. The first can be achieved by assigning $(z,s) := (0,(a+1)/2)$. For the second, we may perturb the current value of z just enough to make it an integer. (The **protecting** clause means that the relation $z\leq\sqrt{a}<z+1$, achieved by the instantiated schema, should not be clobbered when achieving the additional goal $z\in Z$.) This can be done by assigning

if $\lceil z\rceil^2\leq a$ **then** $z := \lceil z\rceil$ **else** $z := \lfloor z\rfloor$ **fi**.

An alternative approach to completing the above program is to insist that $z\in N$ hold *throughout* execution of R_5. This is the avenue pursued, for example, in the "structured programming" derivations in [Dijkstra76] and [Blikle78].

6. DISCUSSION

There are a few problems inherent in the use of analogies for program modification and abstraction. These include "hidden" analogies, "misleading" analogies, "incomplete" analogies, and "overzealous" analogies. Hidden analogies arise when given specifications (of the existing program and desired problem in the case of modification, and of the two or more existing programs in the case of abstraction) that are to be compared with one another have little syntactically in common. Since the pattern-matching ideas that we have employed are syntax-based, when the specifications are not syntactically similar, the underlying analogy would be hidden. In such a situation it is necessary to rephrase the specifications in some equivalent manner that brings their similarity out, before an analogy can be found. This is clearly a difficult problem in its own right; in general some form of "means-end" analysis seems appropriate.

At the opposite extreme, a syntactic analogy may be misleading. The same symbol may appear in the specifications of two programs, yet may play nonanalogous roles in the two programs. Two programs might even have the exact same specifications, but employ totally different methods of solution. Situations such as these would be detected in the course of analyzing the correctness conditions for the abstracted programs.

Knowing how a program was constructed can help avoid overzealously applying transformations to unrelated parts of a program. We have seen how the program derivation also helps complete an analogy between two programs, only part of which was found by a comparison of specifications.

REFERENCES

[Blikle78] Blikle, A., "Towards mathematical structured programming," pp. 9.1-9.19 in *Formal Descriptions of Programming Concepts*, ed. E. J. Neuhold, North-Holland (1978).

[Brown76] Brown, R., *Reasoning by analogy*, Artificial Intelligence Laboratory MIT, Cambridge, MA (Oct. 1976).

[Dershowitz81] Dershowitz, N., "The evolution of programs: Program abstraction and instantiation," *Proc. 5th Intl. Conf. on Software Engineering*, pp. 79-88 (Mar. 1981).

[DershowitzManna75] Dershowitz, N. and Manna, Z., "On automating structured programming," *Colloques IRIA on Proving and Improving Programs*, pp. 167-193 (July 1975).

[DershowitzManna77] Dershowitz, N. and Manna, Z., "The evolution of programs: Automatic program modification," *IEEE Trans. Software Engineering* **SE-3**(6), pp. 377-385 (Nov. 1977).

[DershowitzManna81] Dershowitz, N. and Manna, Z., "Inference rules for program annotation," *IEEE Trans. Software Engineering* **SE-7**(2), pp. 207-222 (Mar. 1981).

[Dijkstra76] Dijkstra, E. W., *A discipline of programming*, Prentice Hall, Englewood Cliffs, NJ (1976).

[FikesHartNilsson72] Fikes, R. E., Hart, P. E., and Nilsson, N. J., "Learning and executing generalized robot plans," *Artificial Intelligence* **3**(4), pp. 251-288 (Winter 1972).

[Floyd71] Floyd, R. W., "Toward interactive design of correct programs," *Proc. Information Processing Cong.*, pp. 7-10 (Aug. 1971).

[Gerhart75] Gerhart, S. L., "Knowledge about programs: A model and case study," *Proc. Intl. Conf. on Reliable Software*, pp. 88-95 (Apr. 1975).

[KatzManna75] Katz, S. M. and Manna, Z., "Towards automatic debugging of programs," *Proc. Intl. Conf. on Reliable Software*, pp. 143-155 (Apr. 1975).

[Kling71] Kling, R. E., *Reasoning by analogy with applications to heuristic problem solving: A case study*, Ph.D. Dissertation, Stanford Univ., Stanford, CA (Aug. 1971).

[MannaWaldinger75] Manna, Z. and Waldinger, R. J., "Knowledge and reasoning in program synthesis," *Artificial Intelligence* **6**(2), pp. 175-208 (Summer 1975).

[McDermott79] McDermott, J., "Learning to use analogies," *Proc. Intl. Joint Conf. on Artificial Intelligence*, pp. 568-576 (Aug. 1979).

[Sagiv76] Sagiv, Y., *A study of the automatic debugging of programs*, Master's thesis, Weizmann Institute of Science, Rehovot, Israel (Aug. 1976).

[Sussman75] Sussman, G. J., *A computer model of skill acquisition*, American Elsevier, New York, NY (1975).

[UlrichMoll77] Ulrich, J. W. and Moll, R., "Program synthesis by analogy," *Proc. ACM Symp. on Artificial Intelligence and Programming Languages*, pp. 22-28 (Aug. 1977).

[Winston80] Winston, P. H., "Learning and reasoning by analogy," *Comm. ACM* **23**(12), pp. 689-703 (Dec. 1980).