Machine Learning, 10, 7-55 (1993)
© 1993 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Synthesis of UNIX Programs Using Derivational Analogy

SANJAY BHANSALI BHANSALI@SUMEX-AIM.STANFORD.EDU
Knowledge Systems Laboratory, Computer Science Department, Stanford University, 701 Welch Road, Palo Alto, CA 94304

MEHDI T. HARANDI HARANDI@CS.UIUC.EDU
Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue, Urbana, IL 61801

Editor: Jack Mostow

Abstract. The feasibility of derivational analogy as a mechanism for improving problem-solving behavior has been shown for a variety of problem domains by several researchers. However, most of the implemented systems have been empirically evaluated in the restricted context of an already supplied base analog or on a few isolated examples. In this paper we describe a derivational analogy based system, APU, that synthesizes UNIX shell scripts from a high-level problem specification. APU uses top-down decomposition of problems, employing a hierarchical planner and a layered knowledge base of rules, and is able to speed up the derivation of programs by using derivational analogy. We assume that the problem specification is encoded in the vocabulary used by the rules. We describe APU's retrieval heuristics that exploit this assumption to automatically retrieve a good analog for a target problem from a case library, as well as its replay algorithm that enables it to effectively reuse the solution of an analogous problem to derive a solution for a new problem. We present experimental results to assess APU's performance, taking into account the cost of retrieving analogs from a sizable case library. We discuss the significance of the results and some of the issues in using derivational analogy to synthesize programs.

Keywords. Derivational analogy, program synthesis, learning, UNIX programming

1. Introduction

Several systems based on the reuse of a design process have been developed in recent years, for domains that include program synthesis (Baxter, 1990; Goldberg, 1990; Mostow & Fisher, 1989; Steier, 1987; Wile, 1983), circuit design (Huhns & Acosta, 1987; Mostow et al., 1989); Steinberg & Mitchell, 1985), mathematical reasoning (Carbonell & Veloso, 1988), physics problems (Hickman & Lovett, 1991), human interface design (Blumenthal, 1990), and blocks-world planning (Kambhampati, 1989; Veloso and Carbonell, 1991). Though these systems have demonstrated the effectiveness of a reuse-based paradigm for improving efficiency in search-intensive tasks, most of them have been tested in the restricted context of an already supplied base analog, or on a few isolated examples, or for "toy" problem-domains. Some of the open issues in using the derivational analogy approach are: Would the approach scale up for complex, real-world problems? How does the cost of retrieving analogs from a sizable episodic memory affect the system's overall performance? How costly is the effect of an inappropriate analog? What, if any, characteristics of the domain, its representation, or the underlying problem-solver determine the effectiveness of the approach?

In this paper, we address the two issues related to the effect of retrieval—effect on overall performance and cost of inappropriate analogs—in the context of a prototype system, APU (*A*utomated *P*rogrammer for *U*nix), that uses derivational analogy to synthesize a program from its specification (Bhansali & Harandi, 1990a; Bhansali & Harandi, 1990b; Harandi & Bhansali, 1989; Bhansali, 1991). Unlike most other implemented systems, APU has the capability to automatically retrieve an appropriate candidate analog from a case library given a new problem specification. Moreover, the search for an appropriate candidate analog can be done at an arbitrary point in the program synthesis process, making it possible for the system to smoothly interleave replay and problem-solving in a manner that is transparent to the user. We describe experimental methodologies to determine whether derivational analogy is a viable mechanism for improving the overall problem-solving performance when the cost of retrieval and the cost of recovering from misapplied analogies are factored in. Following the style of Minton (Minton, 1988), we assess APU's performance on a population of real-world problems generated randomly by fixed procedures. The results of the experiment point to certain criteria that may be used to predict the cost-effectiveness of using derivational analogy.

We restrict ourselves to the specialized domain of synthesizing UNIX shell programs. Even though UNIX programming is similar to conventional programming in some respects, there are several features that are unique to UNIX programming, e.g., no support for data structures, a rich set of highly reusable commands and library routines as well as an interface that permits a compositional style of programming. On the other hand, the shell has all the commonly used control structures of programming languages like conditionals, loops, and sequences, and it has facilities for input-output, subroutines, parameter-passing, as well as recursive calls. This makes the domain sufficiently general so that the approach described here should be extensible to other programming domains.

The rest of the paper is organized as follows: Section 2 gives an overview of the entire system. Section 3 describes how programs are synthesized without using analogy by going through the derivation of an example in detail. Section 4 describes the derivational analogy subsystem using an example of a program derived using analogy. Section 5 presents empirical results to demonstrate the feasibility of the approach. Section 6 discusses related work and some of the properties of our representation scheme that determine the effectiveness of the approach. Finally, Section 7 summarizes the main results of this work.

2. Overview of APU

Figure 1 illustrates the two main components of APU—the knowledge base and the program synthesizer. The knowledge base contains a description of the set of *primitive* objects, functions, and predicates used in problem specification, rules for decomposing problems, and a library of subroutines, cliches (code templates), and derivations of previously solved problems. We assume that problem specifications and rules are encoded using the same vocabulary. Although this assumption seems restrictive, some recent work (Miriyala and Harandi, 1991) suggests that schemas and case-based reasoning techniques can be applied to help users formulate problem specifications in the terminology used to encode a knowledge base. The information in the knowledge base is used by the program synthesizer to transform

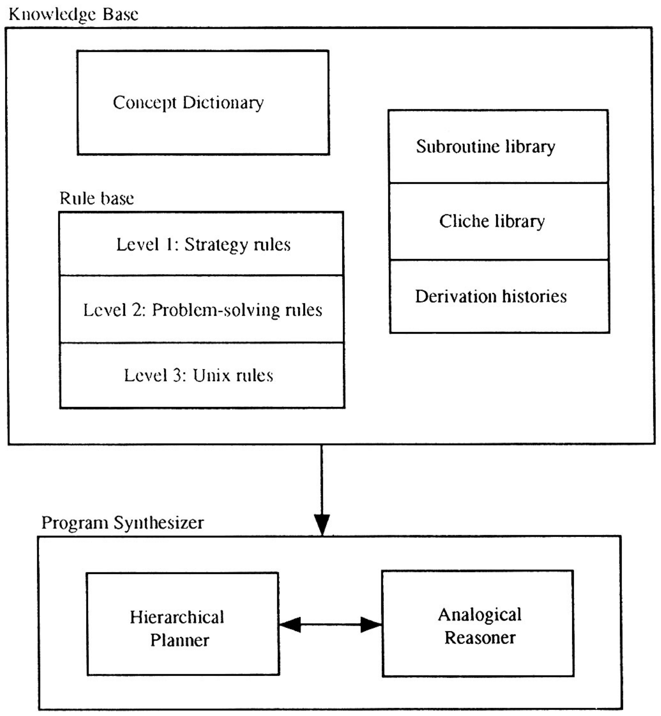


Figure 1. The building blocks of APU.

a user-specified problem to the final code. The program synthesizer consists of two components—a *planner* and an *analogical reasoner.* The planner embodies the control knowledge for choosing a sequence of rules to derive a program from a problem specification. The analogical reasoner works in an integrated manner with the planner; it looks for a problem that is analogous to the current problem being solved, and uses its derivation to derive a solution for the new problem.

2.1. Knowledge base

The knowledge base consists of three major components: a *concept dictionary,* a *library* of components, and a *rule base.*

2.1.1. Concept dictionary

The concept dictionary contains a description of all the objects, predicates, and functions known to the system and provides the ontology for specifying problems and formulating rules. The concepts are represented using a frame-based notation. The representation of objects includes slots for various attributes and the subclass relation between objects. The representation for functions and predicates includes information about the input and output arguments together with default values for them. Figure 2 shows examples of objects, functions, and predicates in APU's concept dictionary. The class relation between concepts defines a hierarchical relationship between objects, functions, and predicates. Figure 3 shows parts of these hierarchies.

The function and predicate description provides their signature (number and type of arguments), but does not fully define their semantics. The semantics of the functions and predicates is operationally defined by the rules used in transforming them. These rules (discussed in the following section) are formulated in terms of the most abstract concepts in the type hierarchies, and are used to transform expressions that contain instances of the abstract concept. These polymorphic rules and the abstraction hierarchies in the concept dictionary form the basis of APU's analogical reasoning discussed in section 4.

```
(defobject :NAME directory-object
           :ISA directory-object
           :WITH (NAME word
                  ACCESS-RIGHTS word
                  OWNER user
                  GROUP word
                  SIZE integer
                  DATE-OF-ACCESS date))

(def-function :NAME Nth-frequent
           :ISA predicate
           :WITH (INPUT (REQUIRED x :object )
                        (REQUIRED y :collection)
                        (REQUIRED KEYWORD Nth :integer)
                        (OPTIONAL KEYWORD KEY :integer
                            (DEFAULT 1))))

(def-function :NAME most-frequent
           :ISA Nth-frequent
           :WITH (INPUT (REQUIRED x :object )
                        (REQUIRED y :collection)
                        (OPTIONAL KEYWORD Nth (ALWAYS 1))
                        (OPTIONAL KEYWORD KEY (DEFAULT 1))
(def-predicate :NAME rel-op
           :ISA predicate
           :WITH (INPUT (REQUIRED x :number)
                        (REQUIRED y :number)
```

Figure 2. Examples of objects, functions, and predicates in APU's concept dictionary.

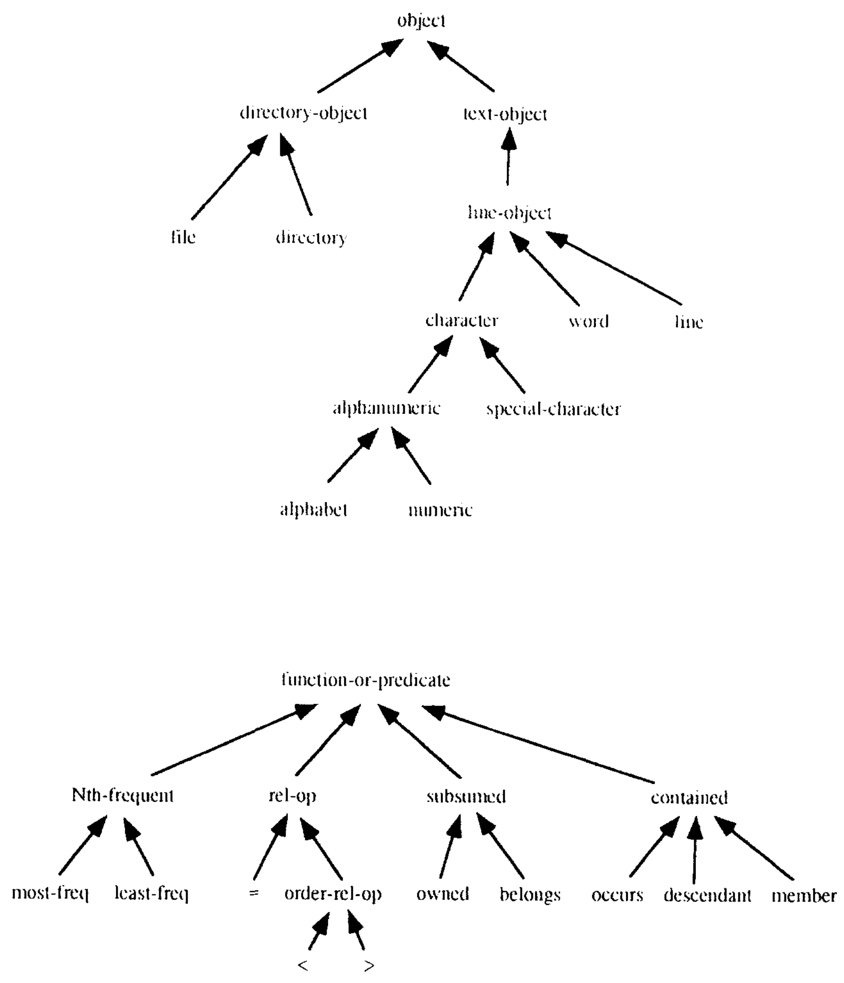


Figure 3. A fragment of the class hierarchy for objects, functions, and predicates in APU.

2.1.2. Rule base

The rule base constitutes the heart of the knowledge base and forms the basis for APU's program-generation capability. The rule base consists of three different levels of rules, each representing a different level of generality. At the top-most level the system has rules dealing with high level *strategies* of problem-solving, which are largely domain independent. Examples of such strategies are the *divide-and-conquer strategy* and the *greedy strategy* (Bhansali, 1991).

At the next level, we have increasingly specialized rules for classes of *problem solving*. The scope of such rules range from general rules that apply to several domains to rules that are specific to a particular domain—in our system, the operating system domain. A simple example of such a rule is:

Rule: *To get the nth-maximum element from a list of elements, sort the list in decreasing order, and select the nth element from the head of the list.*

The third category of rules are the *UNIX* rules, which are specific to the UNIX operating system. These rules describe the function of UNIX commands and subroutines. An example of such a rule is:

Rule: *To list all sub-objects of a directory use command* ls.

The level of a rule provides a rough estimate of its efficacy in finding a solution for a problem as well as of the quality of the resultant program. Generally speaking, a UNIX rule is more effective than a problem-solving rule which in turn is more effective than a strategy rule. Intuitively, this is a reflection of the power versus generality trade-off. However, this is just a heuristic and does not always yield the most efficient program, e.g., there are cases when a combination of two or more commands, obtained by first applying problem-solving rules and then UNIX rules, is more efficient than a single command produced by a single UNIX rule. But, for a majority of the cases, our experience has shown this to be a reasonably good heuristic. This fact becomes important while deriving a program by derivational analogy when we address the issue of appropriateness of replaying a plan step.

Rule representation: Terminology and notation. Formally a rule in our system is a 5-tuple $\langle G, F, B, T, L \rangle$, where G is the *goal* that needs to be solved, F is a *filter* consisting of a set of conditions that must be satisfied for the rule to be applicable, B is the *body* of the rule, T is the *type* of various variables used in the rule, and L is the *level* of the rule. The level of the rule is used to control rule application, which is discussed in section 2.2.

The body of the rule is a program fragment containing a set of three kinds of statements (Dershowitz, 1985):

- **assert** $\alpha(\bar{x})$
- **achieve** $\beta(\bar{y})$
- **P** $(\bar{z})$

where $\alpha(\bar{x})$ and $\beta(\bar{y})$ denote formulae over parameter lists $\bar{x}$ and $\bar{y}$, respectively. The **assert** statement specifies conditions that are true at that point in the program fragment, the **achieve** statement specifies a subgoal that must be solved, and **P** is a piece of 'primitive' code in the target language. The primitive code consists of the common programming language constructs, including *if-then-else*, *while-do*, *for-each*, assignment (:=), sequencing (;), and a parallel construct (||), as well as subroutines and cliches in an available library.

The *context* at a point in the program is the set of conditions that are true at that point. This is computed by walking down the derivation structure of a program, and collecting the set of preconditions and the intermediate assertions. We make the simplifying assumption that an assertion remains true, unless explicitly negated by an assert statement.

For readability, we will represent a transformational rule by specifying only the goal, filter, body, and type, in the form:

$$\begin{array}{l} \text{F:}\ \ \text{G} \\ \hline \qquad\ \ \text{B} \\ \textit{where} \\ \qquad\ \ \text{T} \end{array}$$

Definition. A predicate (function) p_g in a goal expression *matches* a predicate (function) p_r in the goal of a rule if $p_g = p_r$ or p_g is a subclass of p_r in the abstraction hierarchy for predicates (functions).

We distinguish between two types of variables in the rule: schematic variables and ordinary variables. Each variable, schematic or ordinary, is associated with a type which is an object in the object hierarchy. Intuitively, a schematic variable of type T is supposed to represent all ordinary variables of type T' where T' is a subclass of T in the object hierarchy.

Definition. A goal G_r in a rule *matches* a goal expression G_g if there exists a substitution σ such that $G_g = G_r\sigma$ and either (1) the variable types in G_g are subclasses of the corresponding schematic variable types in G_r, or (2) the types of the corresponding variables are the same (for ordinary rule variables).

For example, suppose the rule goal has a subexpression (*size ?l*) where the type of *?l* is *line-object*. If *?l* is a schematic variable, then (*size ?l*) matches a goal subexpression (*size ?w*) where *?w* is a variable of type *word*, but not (*size ?f*) where *?f* is a variable of type *file*, since *word* is a subtype of *line-object*, but *file* is not (see figure 3). If *?l* is an ordinary variable, then (*size ?l*) does not match either (*size ?w*) or (*size ?f*). For the rules given in this paper, all the variables in the rule are schematic variables unless stated otherwise.

A rule can be applied to a subgoal if (1) the subgoal matches the goal in the rule, and (2) the context at the subgoal implies the filter of the rule. Thus, in general, determining the applicability of a rule is a theorem-proving task. Since we do not have an implemented theorem-prover in our system, we apply a rule only when the filter condition is either directly specified as facts in the context, or can be computed using an associated procedure.

A rule application is assumed to preserve correctness in that, if the new subgoals are achieved, the original goal will be achieved. Thus, the final program is guaranteed to be correct with respect to the initial goal specification (modulo the correctness of the rules).

The representation and application of rules will be illustrated through an example derivation in section 3.

2.1.3. Library

The library contains three kinds of entities—*subroutines, cliches,* and *derivation histories*. Cliches are partial programs that represent frequently used program structures (Richter et al., 1989; Waters, 1985). A detailed description of the subroutines and cliches in APU is beyond the scope of this paper and may be found elsewhere (Bhansali, 1991). For the purposes of this paper, the most important component of the library is the derivation history library. The derivation history library consists of a list of problems that have been solved by the system, together with their derivations. The structure of a derivation and its role in replay is discussed in sections 3 and 4.

2.2. Program synthesizer

The first component of the program synthesizer is a planner based on the concept of hierarchical planning (Sacerdoti, 1974; Sacerdoti, 1977; Stefik, 1981), whereby first a partial plan consisting of the high-level goals is constructed, which is then refined into more detailed subplans until a complete plan comprised of a set of primitive operators is obtained.

Thus, the planner works at various levels of abstraction. It uses backward chaining to retrieve rules whose antecedents match the current goal. It associates with each goal a *criticality* value (discussed shortly), which is a measure of how critical the solution of the goal is in solving the overall problem. It first tries to solve all goals with the maximum criticality, which forms the highest *abstraction space.* Only after all subgoals in the highest abstraction space are decomposed to less critical goals does the planner attempt to solve other subgoals. If a particular subgoal cannot be solved or decomposed to simpler subgoals, the planner asks whether the user can solve the subgoal manually. Depending on the user's response the planner either backtracks or generates a partial program.

This strategy is intended to prevent the planner from wasting time on unimportant details and enable it to report failure early if it is unable to produce a good partial solution. Moreover, the plan generation technique ensures that if more knowledge is added to the system after a partial program is developed and its derivation recorded, then it will be able to extend the partial solution without having to start from scratch.

To determine *criticality* of subgoals, APU associates numerical values with subgoals, as done by ABSTRIPS (Sacerdoti, 1974), but the values are determined dynamically and not associated *a priori* with predicates. Two heuristics are used to determine criticality of subgoals. The first heuristic uses the types of rules that are applicable to the subgoal: *a subgoal to which a specific rule is applicable is less critical than one to which only more general rules are applicable.* This is based on the rationale that more specific rules are usually applicable for a smaller range of problems but use more knowledge about a problem-solving situation, and therefore have a better chance of finding a successful plan. On the other hand, a more general rule uses less information about the domain, and therefore has a higher chance of failing to find a successful plan. Therefore, it is better to focus first on subgoals for which specific rules are unavailable.

The second heuristic considers the number of rules applicable to a subgoal: *the more rules are applicable, the less critical is the subgoal.* The following formula is used to compute the criticality of the subgoals:

$$\text{Criticality} = \text{rule-level} + 1/n$$

where *rule-level* is 1 if a *UNIX rule* (which is the most specific rule) is applicable, 2 if a *problem-solving rule* is applicable, and 3 if a *strategy rule* (which is the most general rule) is applicable. *n* is the number of rules applicable. If more than one rule is applicable, the most specific rule is used to determine the rule-level. Notice that this strategy associates the highest *criticality* with subgoals that cannot be solved by any rule ($1/n$ = infinity), which is reasonable since we want to ensure that if a user cannot solve the subgoals that APU cannot solve, then the planner should realize this as soon as possible and try other plans.

The *analogical reasoner* uses derivational analogy to speed up the derivation of programs. It uses a set of heuristics to retrieve a previously solved problem that seems analogous to the current problem. It then tries to replay the derivation of that program in the context of the new problem to synthesize a program faster. Before giving a detailed description of the analogical reasoner we illustrate the derivation of a program in APU without using analogy. We will only outline the main steps in the derivation. The example we present will be used later to illustrate the synthesis of an analogous program using derivational analogy. We pretend in the following derivation that APU always finds the correct rule to decompose a problem, and hence does not need to backtrack. Thus, the derivation really represents an idealized history of the program generation process.

3. Example: Most frequent word in a file

The example is to determine the most frequent word in a file. In APU's specification language, this is expressed as follows:

```
NAME: maxword
INPUT: (?f :file)
OUTPUT: (?w :word)
PRECONDITION: true
POSTCONDITION: (most-frequent ?w (COLLECTION (?x :word) :ST (occurs ?x ?f)))
```

The function *most-frequent* is a primitive predicate in APU's concept dictionary, which takes two required arguments—an object and a collection. The predicate is true if the object is the most frequently occurring one in the collection.

Figure 4 illustrates APU's plan synthesis algorithm. The APU planner always begins by searching the UNIX specific rules to see if there is a direct UNIX command to solve the problem. This enables APU to find more efficient plans in favor of less efficient ones, based on our assumption that programs that use a single library routine for a task are more efficient than (or as efficient as) the ones that use a combination of two or more routines for the same task. If a problem cannot be solved by a direct UNIX command, APU tries to solve the problem by using analogy from a previously solved problem. If no analogs are found, APU looks for rules in the rule base to see if the problem can be decomposed into simpler problems. If it finds such a decomposition, the algorithm is used recursively on each of the subproblems; otherwise the planner reports a FAILURE. For this derivation, we assume that the plan library used by the analogical reasoner is empty, so that the call to the analog retriever always returns nil.

```
SOLVE-USING-PLANNER(P)
begin
    solution := FAILURE;
    if direct-solution(P) then
        solution := compute-direct-solution(P)
    else
        S := source-analogue(P); % source-analogue(P) returns a single source analogue for S or nil
        if S then
            solution := ANALOGY(S,P);
        while (solution == FAILURE) and (applicable-rules(P))) loop
            $p[1...n]$ := apply-rule(P, next-applicable-rule(P));
            for each subproblem $p_i$ do
                $sol_i$ := SOLVE-USING-PLANNER($p_i$);
            solution := compose($sol_1, \ldots, sol_n$)   % if one of the $sol_i$'s is FAILURE, then compose
        endloop                                         % returns FAILURE
    return solution;
end
```

Figure 4. The plan-synthesis algorithm of APU.

For the above problem, APU fails to find an applicable UNIX rule, but finds a problem-solving rule to reduce the problem to three subproblems:

Rule: R1
True: (nth-frequent *?e* (COLLECTION(*?x* :object) :ST $\overline{?constraints}$)
:Nth *?nth* :KEY *?k*)

---

achieve (= *?l* (SET (*?x'* :object *?n* :integer) :ST (∧ $[\overline{?constraints}]_{?x,?x'}$
(= *?n* (count-of *?x'* (COLLECTION(*?x* :object)
:ST $\overline{?constraints}$))))));
achieve (= *?tuple* (Nth-maximum *?l* :Nth *?nth* :KEY last :ORDER >));
achieve (= *?e* (select-field *?k* *?tuple*))

where
(*?e* :object; *?l* :List(object integer); *?tuple* :Tuple(object integer); *?nth*, *?k* :integer)

In the above rule, $\overline{?constraints}$ denotes a set of constraints, and $[\overline{?constraints}]_{?x,?x'}$ denotes the expression $\overline{?constraints}$ in which all free occurrences of *?x* have been replaced by *?x'*. Nth-frequent is an abstract predicate in APU's concept dictionary that takes two mandatory arguments—an object and a collection of objects—and two optional, keyword arguments, denoted by :Nth and :Key, respectively. :Nth specifies the value of N, and :Key specifies which component of the tuple is to be considered (note that in general a collection is a multi-set of tuples). The predicate is true if the object is the :Nth most frequent one in the collection with respect to the :Key component. The default values for both these arguments is 1. 'Last' is a special value which is interpreted to be the last component of a tuple—in this case, 2. Thus, an English paraphrase of the rule says: To find the Nth most frequent object in a collection of objects satisfying certain constraints, first find the set of objects and the number of times each one occurs in the collection (the first *achieve* statement), then find the tuple having the Nth maximum value of the second component of the tuple (the second *achieve* statement), and then select the first component of that tuple (the third *achieve* statement).

The application of this rule results in the following partial program:

achieve (= *?l* (SET (*?w'* :word *?n* :integer) :ST ($\wedge$ (occurs *?w'* *?f*)
                (= *?n* (count-of *?w'* (COLLECTION(*?x* :word) :ST (occurs *?x* *?f*))));
achieve (= *?tuple* (Nth-maximum *?l* :Nth 1 :KEY 2 :ORDER >));
achieve (= *?e* (select-field 1 *?tuple*))

This program is represented in APU's working memory as the derivation tree sketched in figure 5. (Section 4.2. describes how the information in the derivation tree is used during replay.)

The planner now computes the criticality of the three subgoals. It discovers that the first subgoal can be solved by three strategy rules, the second goal by a single problem-solving rule, and the third subgoal by a UNIX rule. Therefore, the criticality assigned to the three subgoals is 3.33 (= 3 + 1/3), 3 (= 2 + 1), and 2 (= 1 + 1), respectively. Consequently, the planner tries to decompose the first subgoal.

The three strategy rules applicable to this subgoal can be paraphrased in English as follows:

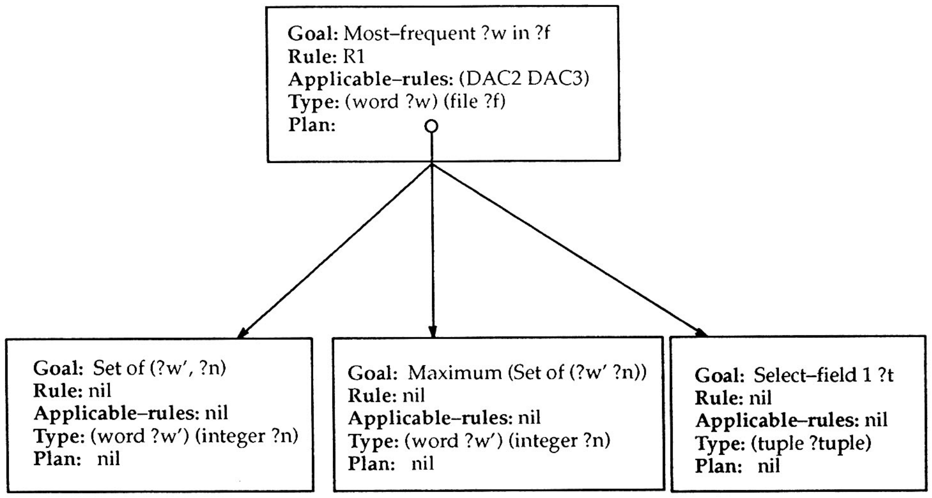


Figure 5. Partial program after application of one rule.

Divide-and-Conquer2 Rule. *To compute a list of objects A_1, A_2 satisfying the relations $R_1(A_1)$, $R_2(A_2)$, and $R_{12}(A_1, A_2)$, compute the list of A_1 satisfying $R_1(A_1)$, compute the list of A_2 such that $R_2(A_2)$ is satisfied, and take a join of the two lists such that $R_{12}(A_1, A_2)$ is satisfied.*

Divide-and-Conquer3 Rule. *To compute a list of objects satisfying constraints ?const1 and ?const2, compute the list of objects satisfying ?const1, compute the list of objects satisfying ?const2, and take the intersection of the two lists.*

Generate-and-Extend Rule. *To compute a list of objects A_1, A_2, generate the list of A_1 satisfying $R_1(A_1)$, and iteratively compute tuples of (A_1, A_2) such that the relation $R_{12}(A_1, A_2)$ is satisfied, until no more tuples can be found.*

The second rule is more general than the first one, since it applies to a list of any objects, whereas the first one only applies to a list of tuples with two components. The third rule is essentially a modification of the first one, where the 'join' operation is done iteratively on each component of the tuple. The first two rules lead to unsuccessful plans since they involve a subplan of enumerating all words and all integers, causing the planner to backtrack. For this example, we will assume that the planner chooses the third rule. The formal specification of the rule is:

Rule: Generate-and-Extend

```
(finite-domain (SET ($?x_1$: object) :SUCH-THAT $\overline{?cond_1}$)):
    (= $?z$ (SET ($?x_1$ :object $?x_2$ :object) :SUCH-THAT ($\overline{?cond_1} \wedge \overline{?cond_2}$)))
------------------------------------------------------------------------
    achieve (= $?y_1$ (SET ($?x_1$ :object) :SUCH-THAT $\overline{?cond_1}$));
    while (read $?v_1$)
    do
        achieve (= $?y_2$ (SET ($?x$ :object) :ST $[\overline{?cond_2}]_{?x_1,?v_1;?x_2,?x}$));
        for $?v_2$ in $?y_2$
        do
            achieve (appended $?v_1$ $?v_2$ :TO $?z$);
        done
    done < $?y_1$
where
    ($?z$ :Set(object object); $?y_1$, $?y_2$ :Set(object); $?v_1$, $?v_2$ :object)
```

where $\overline{?cond_1}$ represents those constraints that contain only $?x_1$ as a free variable and $\overline{?cond_2}$ represents the rest of the constraints. The body of the rule contains a shell-construct (WHILE-DO) that reads each element from a list, does some computation on the object, and outputs the result to a variable. This rule could actually be composed of several low-level rules, e.g., a rule that says that a set of objects may be implemented as a stream with each object in a separate line, a rule that says that a loop construct iterating on successive lines of a file may be implemented using the WHILE-DO construct and the input redirection primitive ('<' on the last line of the rule body), etc. However, for efficiency, it is sometimes preferable to collapse several of these rule applications into one rule (Barstow, 1979).

The application of this rule reduces the first subgoal to the following program fragment:

```
achieve (= ?y1 (SET (?w') :SUCH-THAT (occurs ?w' ?f)))
while (read ?v1)
    achieve (= ?y2 (SET (?n :integer)
                    :ST (= ?n (count-of ?v1 (COLLECTION(?x :word)
                                              :ST (occurs ?x ?f))))
    for ?v2 in ?y2
    do
        achieve (output ?v1 ?v2 > ?z);
    done
done < ?y1
```

A computation of the criticality of the subgoals results in the discovery that there is no rule that can be used to reduce the goal of finding the set of integers[1] representing the number of times a word occurs in a file. This subgoal is assigned a criticality of infinity. The planner informs the user that the subgoal cannot be decomposed further. The user can now ask the planner either to backtrack, or else to continue and produce a partial solution. For this particular subproblem, it is relatively easy to write a small C or Pascal program, whereas writing a shell script using UNIX commands is quite awkward. Therefore, let us assume the user chooses to continue.

The planner then goes ahead to generate the rest of the plan by choosing an applicable rule and reducing a subgoal until all the subgoals are reduced to primitive commands. Figure 6 shows the derivation trace of the complete program, showing the goal and the rule used at each node. The Appendix contains an English paraphrase as well as the formal representation of each rule.

The next stage is concerned with the conversion of the derivation tree into a program. This is done in three passes, using a set of transformational rules, and is described elsewhere (Bhansali, 1991). The final result of the transformations is the program shown below (except for the comments in italics):

```
cat ?f |
tr -s ' ' '\012'|                           # replace spaces by newline
tr -s '     '' \012'|                       # replace tabs by newline
sort |                                      # sort the list
uniq > /tmp/file728                         # remove duplicates
WHILE read ?v370
DO
  [(SET (?x :integer) :ST
    (= ?x (count-of ?v370 (COLLECTION(?x :word) :ST (occurs ?x ?f))))) > ?y2]
  FOR ?v371 IN ?y2
  DO
    echo $?v370 $?v371 > > /tmp/file729     # output word, word-count
  DONE
```

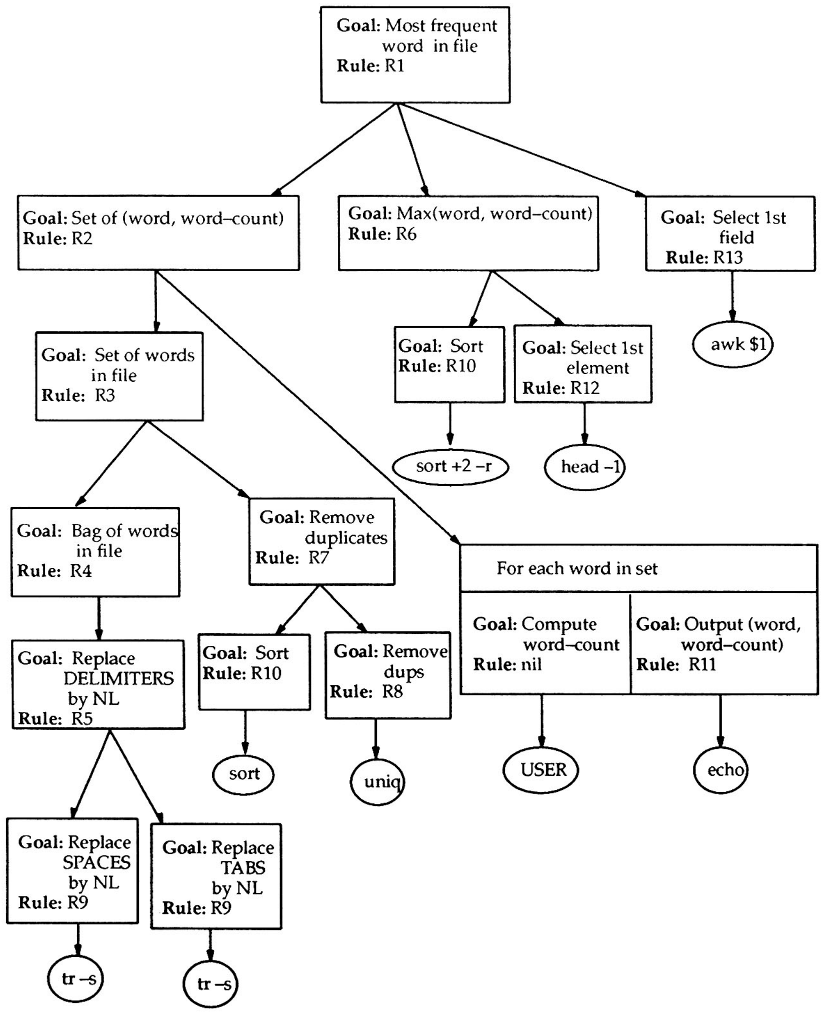


Figure 6. Derivation tree for the program to find the most frequent word in a file.

```
DONE < /tmp/file728                      # the input to the while loop
sort +2 -r /tmp/file729 |                # sort on the second field
head -1 > /tmp/file730;                  # select the first element of the list
set ?w='awk'{print $1}' /tmp/file730'    # select the first field
```

The program is complete except for the computation of the word-count, which has to be supplied by the user.

4. Program derivation using analogy in APU

In the derivation given in the previous section, it was assumed that the planner always chooses the correct rule to decompose a problem. However, more realistically, a planner might spend a lot of its time searching for the correct rules and backtracking, which slows down the program synthesis process. By recording the steps in the derivation of a problem, the planner can reduce search time by replaying portions of an old plan, in the context of a new, analogous problem.

APU's analogical reasoning consists of two main processes: *retrieval* of a source analog from the derivation history library, and *elaboration* or replay of its derivation history in order to solve a target problem. For ease of exposition and maintaining continuity with the previous section, we reverse the order of presentation of these two stages: we first describe how the derivation given in the previous section is used to derive an analogous program, and then describe APU's retrieval mechanism.

4.1. Elaboration: Replay of plans

A derivation of a problem consists of the subgoal structure of the problem showing the decomposition of each goal into its subgoals. With each subgoal that is solved by the planner the following information is stored:

1. The expression representing the subgoal.
2. The subplan used to solve it. This is essentially a pointer to the sequence of subsubgoals into which the subgoal is decomposed (see figure 5). Thus the derivation has a recursive tree-like structure.
3. A pointer to the rule applied to decompose the problem.
4. The set of other applicable rules.
5. The types of the various arguments.
6. The binding of rule variables to subexpressions in the goal.

The subgoal at a particular node forms the basis for determining whether the subplan below it could be replayed in order to derive a solution for a new problem. When a subplan is stored in the derivation history, the subgoal is indexed using the retrieval heuristics to be described in the next section. The decision to store a particular subplan is currently made by the user.

The rules stored at the node contribute to some of the speedup of replay over direct planning in two ways. The first one is based on the rationale that if the rule applied to the source problem is applicable to the target problem, then, since the rule led to a success for the source, it is likely to lead to a success for the target. If there are potentially several rules, and only a few of them lead to success, then this rationale could result in considerable speedup by avoiding rules that led to failure. (Of course, in some cases this could result in degradation of performance by deliberately leading the system to a failure path. But, if the retrieval heuristics are good enough, this should not happen too often.) The set of other applicable rules results in some speedup if the original rule is found to be inapplicable, since the system does not have to recompute the set of applicable rules when they are still valid for the target subgoal.[2]

The types of the variables are useful for determining the best analogs for a given problem by comparing them with the types of the corresponding variables in the new target problem. If the types of all corresponding variables are identical, it may represent a perfect match (depending on whether the corresponding predicates and functions are identical), and the derivation below the node can be copied (after checking that the filter conditions of the rule still hold)—a much faster operation than replay.

Finally, the binding of variables is used to establish correspondence between variables in the target and source. Expressions bound to the same rule variable are assumed to correspond (Mostow et al., 1989; Mostow & Fisher, 1981; Steier, 1987).

Figure 7 gives an outline of the algorithm to derive a program using analogy. We illustrate the working of an algorithm by an example.

```
ANALOGY(S,T)
S is a pointer to the derivation of a source problem (subproblem) and T is the target problem
(subproblem) to be solved. The algorithm returns either a plan for achieving T or FAILURE.
begin
  if applicable-rule(rule(S), T) then
    if identical-subgoals(goal(S),T) then
      solution := copy-subtree(subplan(S))
    else
      $s[1...m]$ := apply-rule(rule(S),T)          % the rule application produces subproblems
      for i := 1 to min(m,n) do                    % $s_1, \ldots, s_m$ for S and $t_1, \ldots, t_n$ for T.
        if has-direct-solution($t_i$) then
          $sol_i$ := compute-direct-solution($t_i$)
        else
          $sol_i$ := ANALOGY($s_i$, $t_i$);              % assume $s_i$ and $t_i$ correspond.
          if $sol_i$ == FAILURE then
            $sol_i$ := SOLVE($t_i$);
      if $n > m$ then
        for i := $m + 1$ to $n$ do
          $sol_i$ := SOLVE($t_i$);
      solution := compose($sol_1, \ldots, sol_n$)
  else
    solution := try-other-rules(S,T)
  return solution;
end
```

Figure 7. The analogy algorithm.

4.1.1. Example: Most frequent file in a directory

The example is to derive a program to find the most frequent filename among descendants of a given directory. This example may seem a bit contrived and has been chosen for illustration purposes only. The specification of the program is:

```
NAME: maxfile
INPUT: (?d :directory-name)
OUTPUT: (?fn :file-name)
PRECONDITION: true
POSTCONDITION: (most-frequent ?fn (COLLECTION(?y :file-name) :ST (descendant ?y ?d)))
```

We assume that APU has already solved the **maxword** problem described in section 3. The top-level algorithm (figure 4), after determining that there is no direct UNIX command to solve the problem, attempts to find an appropriate source analog. We describe in the next section how APU's retrieval heuristics are used to retrieve the *maxword* problem as an analog for the *maxfile* problem. For now, we pretend that the *maxword* program is determined to be the most appropriate analog for this problem. The top-level algorithm now calls the ANALOGY algorithm.

The ANALOGY algorithm first checks to see if the source analog rule, associated with the top-level node of the solution, is applicable to the target problem. There are three possibilities:

1. The rule is applicable, the corresponding subexpressions in the target and source are identical up to variable renaming, and the argument types for the corresponding variables are the same (section 2.1.2). Then the two problems are identical and the entire subtree below the source analog is copied (with the appropriate variable substitutions).
2. The rule is applicable, but the corresponding subexpressions in the target and source are not identical, or their corresponding variables are of different types. Then the analogical reasoner applies the rule to the target problem. In general, this rule application would result in a decomposition of a problem into subproblems $s_1, s_2, \ldots, s_m$ for the source and subproblems $t_1, t_2, \ldots, t_n$ for the target. The algorithm attempts to solve subproblems $t_1 \ldots t_{min(m,n)}$ first, by analogy using subproblems $s_1 \ldots s_{min(m,n)}$, and if any of them remains unsolved, by calling the planner. When $n > m$, the problems $t_{m+1} \ldots t_n$ are also attempted using the planner. If any of the subproblems $t_1 \ldots t_n$ remains unsolved (by both the planner and user), the algorithm returns a FAILURE.
3. The third possibility is that the rule is no longer applicable. The algorithm then checks to see whether any of the other applicable rules stored at the source node is applicable. If any of them are, then they are tried in turn until one of them returns a successful subplan for the problem. If none of the rules result in a complete solution, then the analogy algorithm calls the general planner.

Note that in case 2, it is not necessary that t_i always correspond with s_i, and a more general algorithm would put all the s_i's into a pool, from which the appropriate correspondences would be established by reinvoking the retrieval algorithm. However, this would require the identification phase to be repeated after each replay step, making the algorithm very inefficient. The current scheme allows us to skip this identification phase, the trade-off being in missing some of the analogical correspondences (when the t_i's correspond to s_j's, $i \neq j$). Note also that the subgoals t_1, t_2, . . . are solved in the order determined by the rule, and not using the subgoal ordering determined by the criticality measure. Again this reflects a trade-off between recomputing the subgoal ordering versus the possibility that a critical unsolvable goal is not detected early. We chose not to recompute the subgoal ordering, using the argument that most of the time all the target subgoals would be solvable since all the source subgoals were solved.

The ANALOGY algorithm discovers that the source analog rule R1 (shown in section 3), associated with the top-level node of the source derivation, is applicable to this problem with the following binding of parameters:

$$\{?e = ?f,\ \overline{?constraints} = (\text{descendant } ?y\ ?d),\ ?nth = 1,\ ?key = 1\}$$

This is different from the original parameter bindings, where $\overline{?constraints}$ was bound to *(occurs ?x ?f)*, and hence corresponds to case 2 of the ANALOGY algorithm. The decomposition of the problem results in the following three subgoals:

1. Compute the set {(*file-name*, *file-count*)} of files that are descendants of a directory.
2. Select the tuple with the maximum *file-count* from the set.
3. Select the first field from the tuple.

The algorithm now checks whether the subgoals can be solved directly using a UNIX command. Thus, APU's derivational analogy paradigm is sometimes able to improve upon a previous solution even when it is replaying its derivation. In the above case, APU discovers (as before) that the third subtask can be solved using a UNIX command. For the first and second subtasks, there is no direct solution, and hence the analogy algorithm is called recursively using the corresponding subgoals in the source problem as the source analogs.

For the first subgoal the original rule, R3, applies with a different substitution of parameters, and as before, the algorithm uses it to decompose this subgoal to create the following subgoals:

4. Compute a list of unique filenames that are descendants of directory *?d*.
5. Compute the file-count of a given filename among descendants of directory *?d*.

Next, having checked and found that there are no direct commands to solve either of the two subgoals, the analogical reasoner tries to apply the same rules used in the corresponding source nodes. For subgoal 4, this produces the two subgoals:

6. Compute a list of files that are descendants of directory *?d*.
7. Remove duplicates from the list.

The analogical reasoner continues to reason as before until it comes to the point where it tries to apply rule R4 in the source derivation. At this point it discovers that the original rule used to get a list of words in a file no longer applies for getting a list of files that are descendants of a directory, since that rule is only applicable for line-objects in a stream. Thus, this problem is passed to the planner, which synthesizes the code for finding a list of files that are descendants of a given directory and returns to the analogical reasoner, reporting a success.

Subgoal 5 was unsolved in the source solution, and so the analogical reasoner finds no applicable rule for it. This subgoal is also passed to the planner. The planner (as before) fails to find a solution for it and asks if the user can solve it. As before, we assume that the user answers *Yes*, and the planner returns to the analogical reasoner, reporting a success.

For the second subgoal, the algorithm discovers that the same sequence of rules used in the source analog are applicable, and it replays the subtree below that node in the derivation tree to obtain the same subplan as before.

Figure 8 shows the derivation tree for the target problem. The completed plan is transformed into a program using the transformation rules mentioned earlier. The complete program for the problem (without the code for computing a list of files that are descendants of the input directory) is shown below:

```
⟨code to get list of files that are descendants of ?d⟩|
sort |                                      # sort the list
uniq > /tmp/file801                         # remove duplicates
WHILE read ?v525                            # for each element in the sorted list
DO                                          # compute the count of a particular
  [(SET (?x :integer) :ST                   # filename in a collection of filenames
     (= ?x (count-of ?v1 (COLLECTION(?x :file) :ST (descendant ?x ?d))))) > ?y2]
  FOR ?var526 IN ?y2
  DO
  echo $?v525 $?var526 ≫ /tmp/file802        # print (file, file-count) in a file
DONE < /tmp/file801                         # the input to the while loop
sort +2 -r /tmp/file802 |                   # sort on the second field
head -1 > /tmp/file803;                     # select the first element of the list
set ?f=‘awk ’print $1’ /tmp/file803‘         # select the first field
```

4.2. Retrieval: Determining source analogs

Many analogy-based systems start off with the assumption that the source analog is explicitly given to the system in the form of a specific cue or a specific goal concept (Burstein, 1986; Dershowitz, 1986; Greiner, 1988; Kedar-Cabelli, 1985). However, in our system, no such information is given. The system has to find the appropriate source analog given the target problem.

We have developed a set of four heuristics to detect candidate analogs. Before presenting the heuristics, we state three desiderata for a retrieval algorithm:

Fast: The retrieval algorithm should be fast. (If retrieving analogs takes so long that deriving programs without analogy is faster, there is no point in using analogy.)

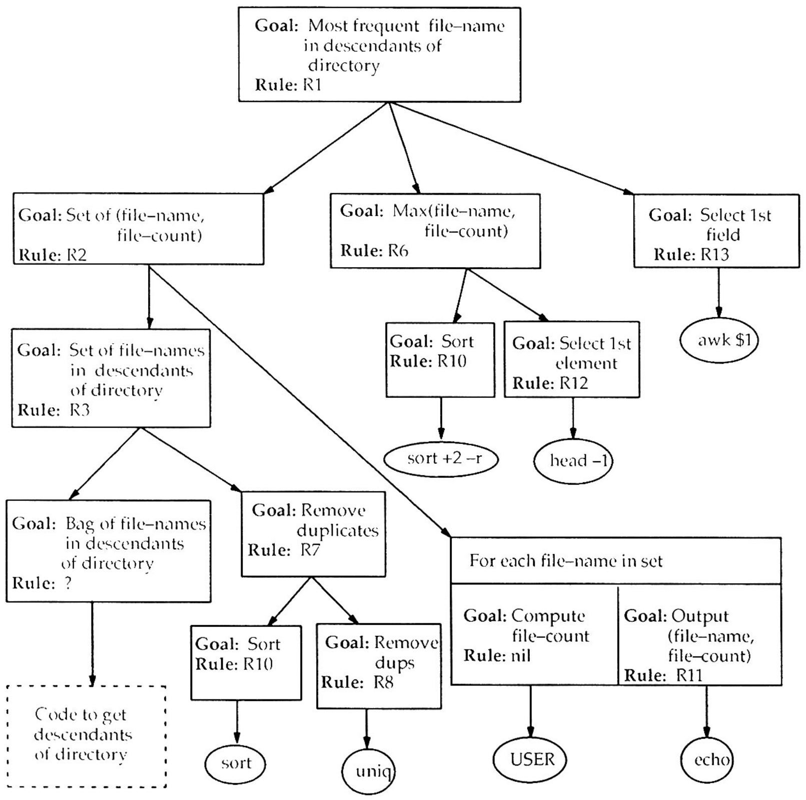

Goal: Most frequent file–name in descendants of directory
Rule: R1
Goal: Set of (file–name, file–count)
Rule: R2
Goal: Max(file–name, file–count)
Rule: R6
Goal: Select 1st field
Rule: R13
awk $1
Goal: Set of file–names in descendants of directory
Rule: R3
Goal: Sort
Rule: R10
Goal: Select 1st element
Rule: R12
sort +2 –r
head –1
Goal: Bag of file–names in descendants of directory
Rule: ?
Goal: Remove duplicates
Rule: R7
For each file–name in set
Goal: Compute file–count
Rule: nil
Goal: Output (file–name, file–count)
Rule: R11
Goal: Sort
Rule: R10
Goal: Remove dups
Rule: R8
Code to get descendants of directory
sort
uniq
USER
echo

Figure 8. Derivation tree for a program to find the most common filename among descendants of a directory. The dashed box represents the program fragment derived without using analogy from the max-word program. (Compare with figure 6.)

Flexible: It is acceptable even if only part of the retrieved analog's solution is applicable to the target problem. (In fact, it is rare that the entire solution for a target problem can be derived by reusing the derivation of a single source analog.)

Best match: If there are several candidate analogs, the retrieval algorithm should select the best analog among them. The best analog is defined as one using which a solution for the target problem can be derived with minimum effort (in terms of CPU time). This is a difficult criterion to satisfy, since there is no operational measure for evaluating the "goodness" of an analog, other than trying all the analogs and measuring their respective times. However, the retrieval algorithm should at least guarantee that if there are several candidate analogs, and one of them is identical to the target problem, then the identical problem should always be retrieved in preference to any other analog since its solution can simply be copied.

1. Solution Structure Heuristic. One way of detecting analogies is to see whether two programs have the same abstract solution structure. The solution structure of a program is determined by the top-level strategies used in decomposing the problem.

Because of the way rules are matched to problem specifications, the top-level rules correspond to the outer-level constructs in a problem specification. Also, the strategy rules consist largely of domain independent rules (section 2.1.2). This suggests that in order to estimate the sequence of top-level strategy rules, one must look at the *domain independent* constructs at the *outermost* level in the problem specification. The domain independent constructs are the various logical and set-theoretic quantifiers, and the logical connectives—*and*, *or*, *not*, etc. Therefore abstract solution structures can be recognized by looking at the parse tree of the specification and extracting the sequence of quantifiers and connectives at the top of the tree.

We illustrate this point with an example. Suppose two problems have the following postconditions:

P1: (NOT (EXIST (?f :file) :SUCH-THAT (and (occurs ?f ?d) (> (size ?f) ?n))))

where ?d is an input directory and ?n is an input integer, and

P2: (NOT (EXIST (?p :process) :SUCH-THAT (and (owned ?p ?u) (> (cpu-time ?p) ?t))))

where ?u is an input user-name and ?t is an input time. The internal representation of the postconditions of both problems after parsing is shown in figure 9. The outermost constructs for both the problems is determined by going down the parse tree until a token is encountered that is not a quantifier or connective. If the token is a predicate it is abstracted as a generic

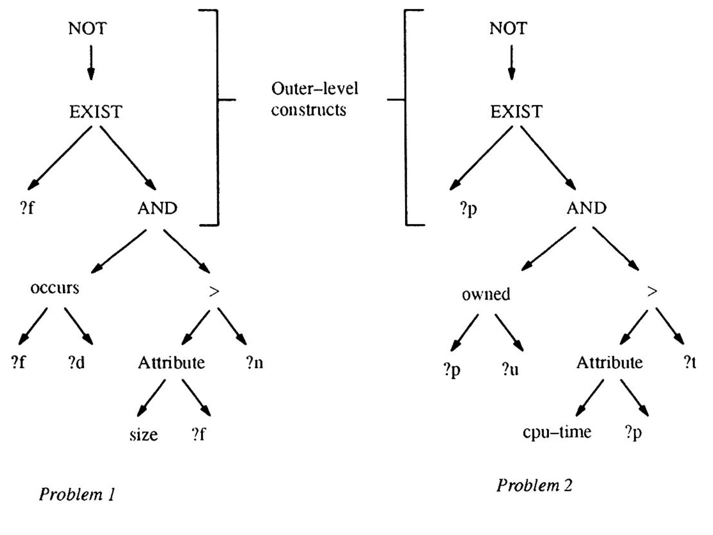


Figure 9. Determining the outermost constructs to identify the abstract solution structure.

?constraint and if it is a variable or constant it is abstracted as a generic variable *?x*. This results in the following outermost construct for both the problems above:

(NOT (EXIST (*?x* : . . .) :SUCH-THAT (AND *?constraint*1 *?constraint*2)))

A postcondition of the form

(NOT (EXIST (*?x* : . . .) :SUCH-THAT (AND *?constraint*1 *?constraint*2)))

is suggestive of a particular strategy for solving problems:

Find all ?x *that satisfy* ?constraint1 *and* ?constraint2 *and delete them.*

Therefore the basic structure of the two problems should be analogous.

Similarly, a program that has a postcondition of the following form:

(= $?z$ (SET ($?x_1$ $?x_2$) :SUCH-THAT *?constraints*))

which describes a set of tuples ($?x_1$, $?x_2$) satisfying the constraints *?constraints*, suggests a *divide-and-conquer* strategy (section 2.1.2):

First form two separate lists of all $?x_1$ and all $?x_2$ satisfying the independent[3] *constraints, and then take a join of the two lists.*

Thus all problems with such a postcondition can be considered analogous.

The other quantifiers and logical connectives result in analogous strategies for writing programs. The solution structure heuristic creates a table of such abstract keys and uses them to index problems. When a new problem is seen, the system computes the structural class to which it belongs and retrieves all problems stored under that class.

2. Systematicity Heuristic. This heuristic is loosely based on the *systematicity principle* proposed by Gentner (1983) and states that: *if the input and output arguments of two problem specifications are parts of a common system of abstract relationship, then the two problems are more likely to be analogous.*

The systematicity principle is a part of Gentner's structure-mapping theory, which describes the set of implicit constraints used in processing analogical mappings. It is based on the intuition that analogies are about relations, rather than simple features. The target objects do not have to resemble their corresponding base objects, but are placed in correspondence due to corresponding roles in a common relational structure.

In our context, the input and output arguments correspond to objects and the predicates and functions relating these arguments correspond to relations. Thus, our heuristic states that for problems to be analogous the input and output arguments have to fulfill analogous roles in a common system of abstract relations.

To implement this heuristic, APU looks at each primitive formula (i.e., not containing the logical quantifiers or connectives) in the postcondition of a problem and forms an abstract *key* for it. The following steps are used in forming the key:

1. Replace all constants by 'Constant'.
2. Replace all input variables by 'input-var'.
3. Replace all other variables, including output variables, by 'var' (deleting the type markers for all the quantified variables).
4. Replace each unary function $(F\ ?x)$ by a binary function $(\mathit{Attribute}\ F\ ?x)$.
5. Replace every predicate (function) by the abstract predicate (function) immediately above it in the abstraction hierarchy (see Section 2.1.1).

Steps 1–3 abstract away the type of a constant or variable. Thus, for example, two expressions $(P\ ?x_1\ C_1)$ and $(P\ ?x_2\ C_2)$, where $?x_1$ is a variable of type T_1, $?x_2$ is a variable of type T_2, and C_1 and C_2 are constants, are considered analogous. The rationale here is that since the same polymorphic predicate P can be used for the different types, then probably the same set of rules would be used to achieve goals involving the two expressions. However, it is still necessary to distinguish between input and other variables and constants since, e.g., the problem of computing $(\mathit{SET}\ (?x_1 : \ldots) : \mathit{ST}\ (P\ ?x_1\ ?x_2))$ where $?x_2$ is an input variable is very different from the problem of computing $(\mathit{SET}\ (?x_2 : \ldots) : \mathit{ST}\ (P\ ?x_1\ ?x_2)$ where $?x_1$ is an input variable. (As a concrete example, consider the problem of finding all files containing a given word versus the problem of finding all words contained in a given file.)

Step 4 abstracts the identity of unary functions by viewing them as an abstract binary function *Attribute* that takes two arguments—the name of the unary function and its parameter—and applies its first argument to the second. The name of the unary function is treated as a constant and ignored (using step 1). In effect, all unary functions are considered 'analogous'. The rationale for this is that in the UNIX domain, most unary functions are attributes associated with an object which are all represented and accessed using similar methods. For example, for a file *?f*, *(size ?f)*, *(owner ?f)*, *(access-code ?f)*, etc. can all be computed by listing all attributes of a file and selecting the desired one (there are exceptions to this heuristic; e.g., *(parent ?f)* cannot be computed in this manner).

The fifth step abstracts higher-order[4] functions and predicates by climbing one step up the abstraction hierarchy. The rationale here lies in the formulation of the rules. Since APU's rules are written in terms of abstract functions and predicates, in order to determine whether the same sequence of rules is applicable to two problem instances, we need to abstract the actual predicates and functions in the problem statement before comparing them. An important issue here is to determine the right level of abstraction or generality: under-generalization would result in missed analogies, whereas over-generalization would result in incorrect analogies. We have adopted a conservative strategy of abstracting only one level, based on empirical studies of several examples in the UNIX domain.

The detection of the higher order relations also establishes the correspondences between the input/output variables of the source and target problem, which is used to prune the set of plausible candidates (using the *conceptual distance heuristic*, discussed shortly) as well as in replay.

The application of the systematicity heuristic for the *maxfile* problem results in the formation of the following keys for the two primitive conjuncts in the postcondition:

```
1. (Nth-frequent var (COLLECTION (var) :ST (contained var input-var)
                     :Nth Constant :KEY Constant)
2. (contained var input-var)
```

The keys are formed as follows: First APU retrieves all the primitive formulae, which consist of (*most-frequent ?f (COLLECTION* . . .)) and (*descendant ?f ?d*). It abstracts the variables and constants, and then replaces the predicates *descendant* and *most-frequent* by climbing one step up the abstraction hierarchy. This results in the formation of the keys (*contained var input-var*) for (*descendant ?f ?d*) and (*Nth-frequent var* (*COLLECTION* (*var*) *:ST* (*contained var input-var*)) *:Nth 1 :KEY 1*), where the values for the keywords *:Nth* and *:KEY* are obtained from the definition of *most-frequent* in the concept dictionary. Replacing all the constants by 'Constant' results in the keys shown above.

To detect analogs using the systematicity heuristic, the system forms a set of keys for each primitive formula in the postcondition of a problem, and indexes the problem with each of those keys. When a new problem is encountered, a set of keys is computed for it and used to retrieve all problems indexed with those keys. The *maxword* problem is indexed under both the keys that were derived for the *maxfile problem* and is selected as one of the source analogs during retrieval. The bindings of the variables in the two keys (considering each *var* and *input-var* in the key as a distinct variable) establishes the correspondence between the following variables in the *maxword* and *maxfile* problem:

$$\{?f = ?d,\ ?w = ?fn,\ ?x = ?y\}$$

Some care has to be taken when forming keys for predicates or functions that are commutative or have a *commutative-dual*, defined as follows:

Definition: Let f and g be two binary predicates or functions. g is a commutative-dual of f if for all x and y, $f(x, y) = g(y, x)$.

Thus, the predicate $>$ is a commutative-dual of the predicate $\leq$, and vice versa. If all specializations of an abstract predicate (or function) in the abstraction hierarchy (section 2.1.1) are commutative or have a commutative-dual, then the abstract predicate (function) is termed commutative. While forming keys, we need to ensure that for commutative operators, the key is not sensitive to the order of the arguments. For example, in problem P2 above, the second constraint could have been written as:

$$(>\ ?t\ (\text{cpu-time}\ ?p))\ \text{or}\ (\leq\ ?t\ (\text{cpu-time}\ ?p))$$

This does not change the essential nature of the problem, and we want to consider all such constraints analogous. Therefore we define a *canonical form* to represent predicates, using the order of a predicate. The canonical form is determined by permuting the arguments of all commutative predicates so that they appear in decreasing order (with variables preceding constants). Thus, the canonical form for the above predicate is:

(Rel-op (Attribute Constant var) input-var)

(where (*Attribute Constant var*) is obtained as an abstraction of (*Attribute cpu-time ?p*)). The system first converts all keys to a canonical form before using them for storing or retrieving problems.

3. Similar Syntactic Feature Heuristic. The solution structure heuristic seeks to detect similar solution structures by recognizing similar patterns in the outer-level (specification) language constructs in the source and target problems. The *similar syntactic feature* heuristic is a closely related heuristic that looks at individual domain independent features in the formulation of problems. Thus, e.g., instead of forming a schema like (*NOT* (*EXIST* (*?x*) *:ST* (*AND ?constraints*))) to index a problem, it might index it using the individual keys *NOT*, *EXIST* and *AND*.

In order to be useful, however, only certain special features in problem specifications, which strongly influence the solution structure, should be considered. We use two classes of problem features that we have found useful for indexing problems. The first one is based on the form of a problem: if the particular form in which the problem is specified indicates the form of the solution, then that problem form should be used to index problems. A particularly useful and commonly occurring form of specification is *recursive* specification. In the UNIX domain, recursive problems can be solved using a shell script written in a file; the shell script has a command that executes the file containing it, and the recursion is implemented by executing this command. Note that this recursive feature of problem specification could not be captured by the systematicity heuristic. For example, consider two problems F_1 and F_2 whose inputs are $?x_1$ and $?x_2$ respectively, and whose outputs are $?z_1$ and $?z_2$ respectively. Let the postconditions of the two problems be:

(AND (IMPLIES (P $?x_1$) (Q (g $?x_1$) $?z_1$)) . . .)

and

(AND (IMPLIES (P $?x_2$) (Q (F_2 $?x_1$) $?z_1$)) . . .)

respectively, where P and Q are predicates and g is a function. The systematicity heuristic would consider both these postconditions analogous, whereas the solutions (shell scripts) for both the problems would be quite different: the shell script for F_2 would have to be encoded in a file and made executable, and the recursive specification would have to be transformed to a command to execute that file. This heuristic is implemented by detecting the occurrence of the name of the problem in the specification, and indexing it as a recursive problem.

The second class of problem features are derived from certain predetermined language keywords. An example of this class of features is the following expression:

(WHEN *?condition ?formula*)

which specifies that when the expression denoted by *?condition* becomes true, the expression denoted by *?formula* must be made true. This expression is usually associated with the following solution structure:

```
loop
    if ⟨code to test ?condition⟩
    then exit;
    sleep;
endloop
⟨code to achieve ?formula⟩
```

and hence all problems having an expression of the above form in the postcondition are indexed with the keyword WHEN.

It should be remarked that this heuristic, by itself, is quite weak in detecting analogous problems. Its utility lies in pruning the set of candidate analogs retrieved by using the first two heuristics.

4. Conceptual Distance Heuristic. This heuristic uses the abstraction hierarchy of objects to determine how "close" the corresponding objects in two problem specifications are. Closeness is measured in terms of the number of links separating the objects in the concept dictionary—the smaller the number of links, the better are the chances that the two problems will have analogous solutions. For example, *lines* and *words* are closer to each other than, say, to a *process*. Therefore, the problem of counting lines in a file is closer to the problem of counting words in a file than to the problem of counting processes in the system.

The closeness between two problems, S and T, is denoted by $dist(S, T)$ and is defined to be the product of the distance between the corresponding objects (determined using the systematicity heuristic) in the two problems. For technical reasons, the distance between two objects of the same type is defined to be 1; the distance between all other objects is 1 plus the number of links separating them. Using this measure and the variable bindings given earlier, the closeness between the maxword and maxfile problems is:

$$\begin{aligned} \text{dist(maxword, maxfile)} &= \text{dist(file, directory)} * \text{dist(word, file-name)} * \text{dist(word, file-name)} \\ &= 3 * 3 * 3 \\ &= 27 \end{aligned}$$

If there is another analog in the library with a smaller closeness measure (e.g., a program to find the most frequent directory name under a directory), then that would be picked as a better analog.

4.3. How the heuristics are combined

In general each of the above heuristics will suggest several, and possibly different, problems as a potential analog of the target problem. The algorithm used by the analog retriever works by retrieving all analogs using the systematicity heuristic and choosing the one that has the maximum number of keys pointing to it. If there is only one such analog, it is returned as the best match. If no, or more than one, analog is found, then the solution structure heuristic is used to select those candidates that share similar abstract schemas in the problem specifications. Further ties are broken by using the syntactic feature heuristic.

If there is no analog retrieved using the first three heuristics, the retrieval algorithm returns a failure. On the other hand, if multiple analogs remain after using all three heuristics, the conceptual distance heuristic is used to select the source analog whose input and output arguments are closest to the input and output arguments, respectively, of the target problem using the ISA hierarchy of objects (see figure 3). If there are still multiple analogs, one of them is returned arbitrarily.

5. Performance results

APU has been used to synthesize 45 different programs, including the *maxword* and *maxfile* programs. The main motivation in this work has been to explore the role of derivational analogy in improving the program synthesis capability of APU. To test the feasibility of the approach, two basic hypotheses need to be investigated:

- Automatic determination of good analogs is feasible.
- Using analogy speeds up program synthesis.

To establish the above claims, we need empirical evidence to assess APU's performance. In this section we describe experiments designed to measure the following aspects of APU's retrieval and replay techniques:

- How good are the heuristics at determining appropriate base analogs?
- How does the time taken to synthesize programs using analogy compare with the time taken to synthesize programs without analogy?
- How does the time taken to synthesize programs depend on the heuristics?
- How does the time for retrieving analogs depend on the size of the derivation history library?

Before presenting the experiments, we describe the methodology for constructing the data set. It must be noted that it is not enough to show results on isolated examples; the system must be tested on a population of problems that is representative of real-world problems. However, the limited knowledge-base of our prototype system precluded testing on a truly representative sample of the space of UNIX programs. Therefore, we decided to restrict ourselves to a subset of the problem domain, consisting of file and process manipulation programs. Problems were constructed randomly from this subset using fixed procedures.

5.1. Generating the data set

We began by constructing a rule base for 8 problems that are typical of the kind of problems solved using shell scripts in this domain. The problems included in the set were:

- List all descendant files of a directory.
- Find most/least frequent word in a file.
- Count all files, satisfying certain given constraints, in a directory.
- List duplicate files under a directory.
- Generate an index for a manuscript.
- Delete processes with certain characteristics.
- Search for given words in a file.
- List all the ancestors of a file.

To generate the sample set, we first created a list of various high-level operations that can be used to describe the top-level functionality of each of the above problems—*count*, *list*, *delete*, etc.—and a list of objects that can occur as arguments to the above operations—*directory*, *file*, *system*, *line*, *word*, etc. Then we created another list of the predicates and functions in our concept dictionary which relate these objects, e.g., *occurs*, *owned*, *descendant*, *size*, *cpu-time*, *word-length*, *line-number*, etc.

Next, we used the definitions of the top-level predicates in the concept dictionary to generate all legal combinations of operations and argument types. For example, for the *count* predicate (which takes two arguments, such that the first one is contained in the second one), the following instances were generated:

```
(count directory system)   ; count directories on the system
(count file system)        ; count files on the system
(count file directory)     ; count files under a directory
(count word file)          ; count words in a file
(count character file)     ; count characters in a file
(count line file)          ; count lines in a file
(count string file)        : count strings in a file
(count process system)     ; count processes in a system
```

In a similar fashion, a list of all legal constraints were generated, using the second list of predicates and functions. Examples of constraints generated include:

- (occurs file directory)
- (descendant directory directory)
- (= int (line-number word file))
- (= string (owner file))
- (occurs character word)

where each argument denotes a variable of that type, which may be either an input or output to a problem specification (see below). Constraints that were trivial or uninteresting were pruned away, e.g., (= *int int*).

Next, we combined these constraints with the top-level operations to create a base set of problems. We restricted each problem to have a maximum of three conjunctive constraints. From this set of about 140 problems, a random number generator was used to select 37 problems which, together with our initial set of 8 problems, formed our sample population of 45 problems.

All the above steps were performed automatically using a small set of simple routines. The final step consisted of translating the high-level description of the problems into a formal specification. This was done manually, in a fairly mechanical manner. The only non-mechanical step was in assigning the input and output arguments for each program. This was done using the most 'natural' or likely formulation of the problem. For example, for the problem (*most-frequent word file*), the corresponding postcondition with a word being the output variable and a file being the input variable is more likely, rather than the reverse (which would generate a program to find that file in which the given input word is the most frequent one).

5.2. *Experiment 1: Feasibility of automatic retrieval*

We stored the 15 randomly chosen problems from the sample set in the derivation history library. Then, for each of the 45 problems, we ran the retrieval algorithm to determine the best base analog. This was done for various combinations of the heuristics. Note that for 15 of the 45 problems, an exact match was available. This was deliberately done, since one of the tests for the retrieval heuristics was that they be able to retrieve an exact match over any other analog. This was not always the case, e.g., when the conceptual distance heuristic was not used.

To evaluate the heuristics, we compared APU's choice against a human expert's, namely the first author. To ensure that our choices were not biased by APU's, we compiled our own list of the best analogs for each problem, *before* running APU's retrieval algorithm. Our criterion for selection was based on writing shell scripts for each problem and evaluating their closeness in terms of the number of common commands and common shell constructs. For some problems, where it seemed that two or more analogs were equally good, we included all the choices, the idea being that if APU's choice matched any of these, it would be considered acceptable.

The result of the experiment is summarized in Figure 10. The first column shows which heuristics were turned on during the experiment. The combinations tried were: all heuristics working, all but one heuristic working, and each heuristic working separately.[5]

The second column shows the number of problems for which APU's choice did not match ours. However, it would not be fair to judge APU's performance simply on the number of mismatches, since that would imply that the human choices are always the best. Since we could not be confident of the latter, after obtaining the mismatches, we again carefully compared APU's choices against ours to judge their respective merits in terms of the quality of the solution or the time taken to derive the solution. We discovered that, in a few instances, APU's choices were clearly inferior to ours, while in others, it was not clear which of the mismatched choices was better. The former were marked as inferior choices (column 3), and an overall score for each heuristic combination was determined by subtracting the number of the inferior choices from the total number of problems (column 4).

5.2.1. *Discussion*

The experiment indicates that using all 4 heuristics, APU's retrieval algorithm performed almost as well as a human (we ignore retrieval time for this experiment). There were only

Heuristics used	# Mismatch	# Inferior solutions	Overall score
All	8	2	43
H_1, H_2, H_3	9	3	42
H_1, H_2, H_4	8	2	43
H_1, H_3, H_4	9	7	38
H_2, H_3, H_4	11	5	40
H_1	13	11	34
H_2	14	8	37
H_3	41	41	4

H_1 - Solution Structure heuristic
H_2 - Systematicity heuristic
H_3 - Syntactic Feature heuristic
H_4 - Conceptual Distance heuristic

Figure 10. Performance of APU's retrieval heuristics against a human expert's.

two cases in which APU's choice of an analog was clearly inferior. The reason why APU failed to pick the correct choice for the two cases became obvious when we looked at the two cases.

Consider the first case, which was to delete all directories that are descendants of a particular subdirectory. This was specified using the postcondition

(NOT (EXIST (?sd: directory) :ST (descendant ?sd ?d)))

where *?d* is an input directory-variable and (*descendant ?sd ?d*) is defined to be true if *?sd* is a subdirectory of *?d* or it is a descendant of a subdirectory of *?d*. The best analog for this problem was the problem of listing all the descendant subdirectories of a directory, specified using the postcondition

(= ?l (SET (?sd: directory) :ST (descendant ?sd ?d)))

since both of them involve recursive traversal of the directory structure in UNIX. However, the analog picked by APU was the problem of deleting all files under a given directory, specified with the postcondition:

(NOT (EXIST (?f: file) :ST (occurs ?f ?d)))

where *?d* is again an input directory-variable. The reason APU picked this analogy was because *occurs* and *descendant* are grouped under a common abstraction *contained* in APU's concept dictionary. Thus, the systematicity heuristic abstracted both *occurs* and *descendant* to (*contained VAR INPUT-VAR*), and considered both to be equally good analogs for the target; the solution-structure heuristic then picked the *delete-files* problem because its outer-level constructs were closer to the target's.

At a more abstract level, APU's inability to pick the right analog can be explained by the fact that APU's estimation of the closeness of two problems in the implementation domain is based solely on its assessment of their closeness in the specification domain. A better organization of the concept dictionary, so that the distance between concepts in the specification domain reflects the corresponding distance in the implementation domain, might avoid some of these missed analogies. For the above example, *occurs* and *descendant* should not be grouped under a common abstraction, since the program fragments for computing the occurrence and descendant relations are very different in the UNIX operating system domain.

The experiment also shows that H_1 and H_2 are the two most important heuristics—as expected. Rows 4 and 5 show the number of missed analogs when one of the two is turned off. Though the table doesn't show it, the problems for which the analogies were missed were also different, indicating that neither heuristic is redundant.

The result in Row 2 was unexpected, since it seems to indicate that the conceptual distance heuristic is unimportant. This was contrary to our experience when we tried the heuristics on isolated examples. In particular, when the base set of analogs contained several similar analogs, the argument abstraction heuristic was important to select the closest one. The reason we got this result is because of the small set of base analogs—there weren't two analogs sufficiently close as to be indistinguishable without using the conceptual distance heuristic.

Finally, H_3 doesn't seem to contribute much to the effectiveness of retrieval. This is again due to the nature of the sample space, where most problem descriptions did not have syntactic cues like keywords and recursion.

5.3. *Experiment 2: Speedup using derivational analogy*

For this experiment, we selected 10 examples at random from our sample set to form the set of source analogs. From the same sample set, we selected another set of 10 examples at random (note that the two sets are not necessarily disjoint) and measured the times taken to synthesize a program for each of them, once with the analogical reasoner off, and once with the analogical reasoner turned on. In order to eliminate user interaction, we tuned APU so that it did not backtrack, and thus the first (possibly partial) solution is accepted for each problem. The experiment was repeated with 20 different sets of base analogs. Figure 11 shows the result of one typical run.

5.3.1. Discussion

Speedup using derivational analogy: The results in figure 12 show that using derivational analogy, the average time to synthesize programs is reduced by almost half, when all the heuristics are used for retrieval.[6] This is not as great as we had expected based on our experience on isolated examples. Nevertheless, the result demonstrates that derivational analogy can improve problem-solving performance, not only on isolated examples, but on populations of problems too.

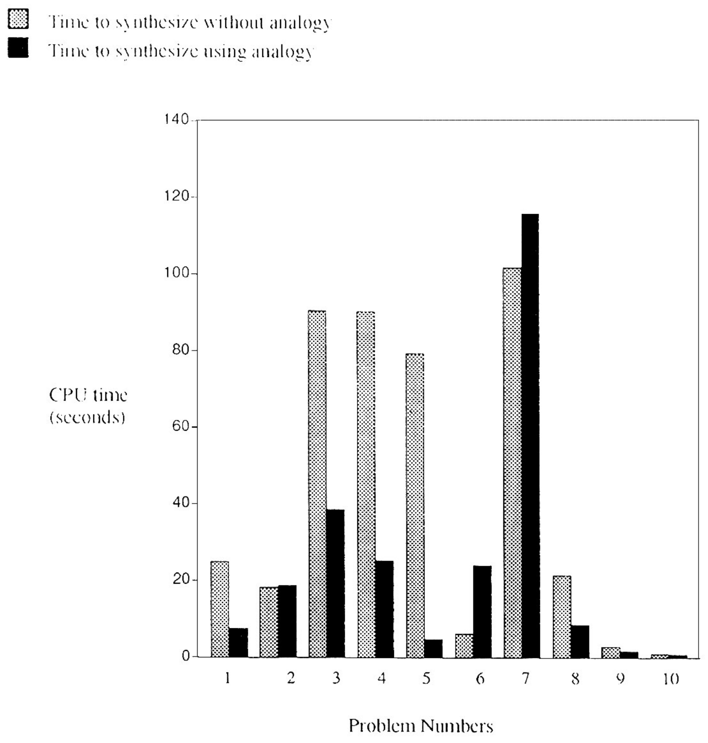


Figure 11. Sample data showing the speedup of program synthesis using derivational analogy.

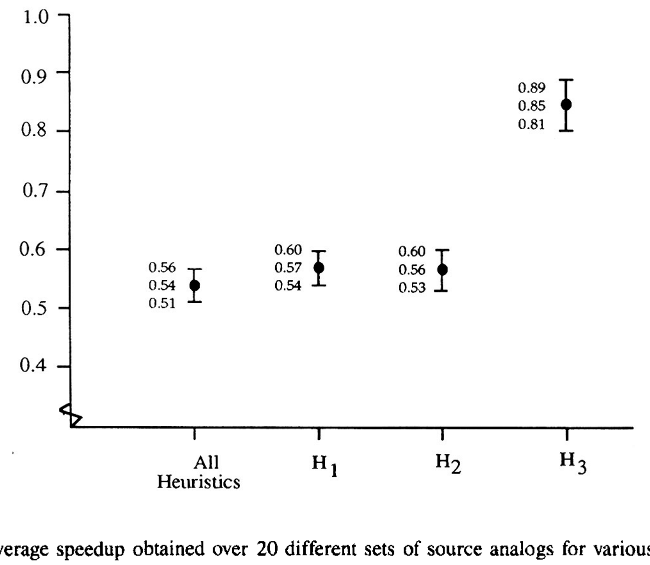


Figure 12. The average speedup obtained over 20 different sets of source analogs for various combinations of retrieval heuristics. The figure shows the average as well as the 95% confidence interval for each value.

There are several factors that affect the generality of these results. First, it is based on the assumption that problems are drawn from the set of basic concepts and constraints with a uniform distribution. However, in practice, we expect a small set of concepts and constraints to account for a large share of the problems encountered in real life. In that case, with a judicious choice of which problems to store in the derivation history library, the number of problems for which close analogs can be found will be much larger than the number of problems without analogs. Consequently, the benefits of using derivational analogy would increase.

Currently, APU's rule base is fairly small, and consists mostly of specialized, domain-oriented rules. As a result, the planner does not spend much time in backtracking during the initial plan synthesis. Moreover, since we tuned APU to eliminate backtracking during the experiment, APU does not achieve much speedup by eliminating search. More realistically, if the rule base had a large number of high-level, general rules, the potential for further speedup by eliminating search would increase. Note that with a large number of general rules the space of problems solvable by the system also becomes large. It is then possible that no problem in the library is close enough to target problems, and using derivational analogy *degrades* the system's performance. Again, we expect that in practice the 80–20 rule would apply, i.e., 80% of the problems would be generated from 20% of the domain concepts, implying that there would be a large number of problems that are analogous, compared to non-analogous ones.

The experimental data also suggest that when target problems do not match analogs in the library, the degradation in performance is small (problems 2, 6, and 7 in figure 11) compared to the improvement in performance when problems match (problems 4 and 5 in figure 11). This suggests that unless the number of mismatches is much larger than the number of matches, derivational analogy would speed up the overall problem-solving. There were also cases (not shown in figure 11) when problems are almost solvable by analogy, i.e., analogies are close but misleading, causing a much more severe degradation in performance. However, this did not happen often enough to significantly affect the overall speedup. Section 6.2 discusses some of the properties of the domain representation explaining this phenomenon.

Finally, the speedup obtained for larger problems (i.e., problems requiring more CPU time to solve without analogy) is generally greater than speedup for smaller problems. This is due to the overheads of matching analogies—for smaller problems the time to retrieve analogs outweighs any speedup that can be gained, even if there is a very close match with a source problem. This could be used to decide when it would be advantageous to use replay, provided an estimation of the size of a user-specified problem is available. Some of the heuristic measures that may be used to estimate the size of a problem include the number of conjuncts in the postcondition, the length of the problem formulation, the degree of abstraction of terms used in the specification language, etc. However, in general, estimating problem difficulty from specifications is difficult and the effectiveness of any heuristic measure would depend on how well correlated it is with the actual problem difficulty.

Effect of heuristics on performance: Experiment 1 (Section 5.2) was designed to measure how good the retrieval heuristics were in retrieving the best analogs; it was found that H_1 and H_2 alone fared roughly 80% to 85% worse than the combination of all four heuristics in finding the best analog. In this experiment we wanted to investigate how a non-optimal choice of the source analog affected APU's performance.

The results show that although H_3 alone was clearly inadequate, using H_1 or H_2 alone, APU fared only slightly worse than when it was using all the four heuristics!

One reasonable conclusion that can be drawn is that the combination of all four heuristics is essentially equivalent to using just one of H_1 or H_2, and the observed small difference is simply due to chance. A Wilcoxon rank-sum test for paired experiments[7] performed on the pairs of experiments (all heuristics, only H_1) and (all heuristics, only H_2) show that at the 1% significance level (i.e., with 99% confidence), H_1 and H_2 individually are not as effective as all the four heuristics. However, though the Wilcoxon test shows that the difference in performance is not due to chance, the *magnitude* of the difference still seems too small to justify using all four heuristics.

One factor that has been ignored in this experiment is the quality of the solution. Notice, that since we do not allow APU to backtrack and accept the first (possibly partial) solution generated, the cost of choosing a wrong analog is not too severe as far as CPU-time is concerned. In the worst case, a subproblem that was solved in the original problem is passed back to the planner and is found to be unsolvable; the planner then returns a failure causing the analogical reasoner to fail at a higher level, and so on, till ultimately, the analogical reasoner fails at the topmost level. The planner then has to solve the problem from scratch. More often, however, subproblems passed to the planner either are solved by the planner or correspond to subproblems that were unsolved in the original problem. In such cases the resultant solution may not be as good as the original one (we assume that the planner always produces the 'best' solution for any problem given to it). Checking a few problems at random in the experiment, we noticed that there were a small number of cases when a solution using all four heuristics was better than that obtained by using just H_1 or H_2. A typical example we found was for the following target problem:

Count all subdirectories under a given directory.

Using all 4 heuristics, the following solution was obtained:

```
ls -l | grep '^d' | wc -l
```

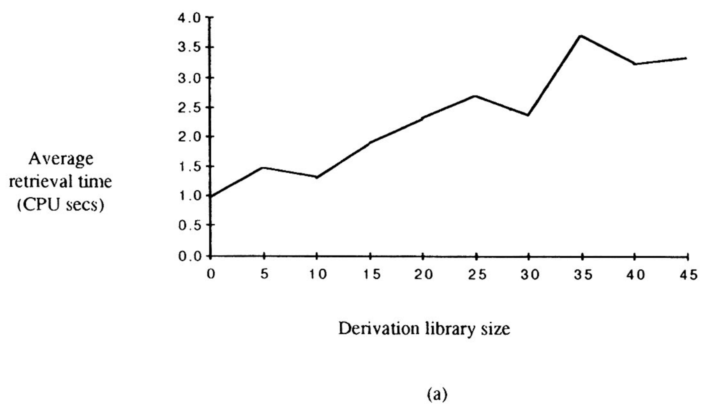
which prints the contents of a directory, selects lines starting with the character 'd' and counts the number of resultant lines. On the other hand, using H_1 and H_2, an analogy from a different program was used and a partial solution was obtained:

```
ls | awk '{⟨test-for-directory⟩}' | wc -l
```

which is clearly inferior to the previous one. If the quality of the solution is also factored into the experiment, so that APU generates the best solution it can, we expect the cost of a wrong analog to be much more severe.

5.4. Experiment 3: Retrieval time

Our third experiment was designed to measure the cost of retrieving analogs as a function of the size of the derivation history library. To measure this, we incrementally increased



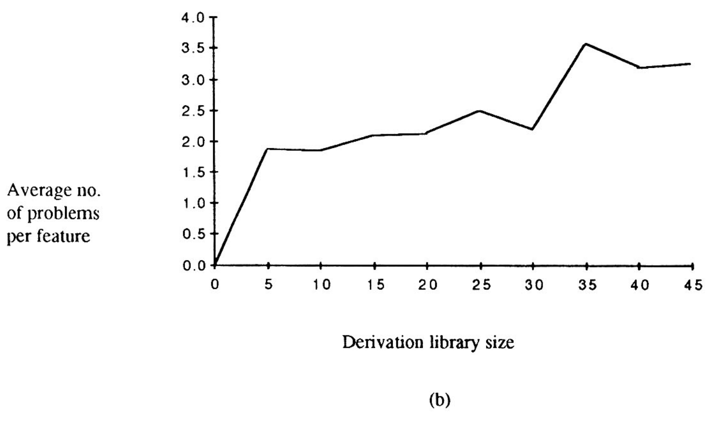


Figure 13. (a) The average time to retrieve analogs as a function of library size. (b) The average number of problems per feature as a funciton of library size.

the size of the derivation history library (in steps of 5, selecting new problems at random), and measured the time taken to retrieve analogs for all the 45 problems. Figure 13 shows the result of one typical run of this experiment.

5.4.1. Discussion

The figure shows that the retrieval time increases almost linearly with the number of problems in the library. This was an unexpected result. Since we index all the problems in the library, based on various features, we expected the retrieval time to grow much more slowly as the size of the library increased.

An examination of the various indices after each problem set is added to the library provided us with an explanation. It must be realized that the time taken to search for analogs essentially depends on the average number of problems indexed on each feature used by the retrieval heuristics. For the retrieval time to converge, the average number of problems per feature should approach a constant value. For our sample set, we found that this was not true. The average number of problems per feature after each set of 5 problems were added to the library is plotted graphically in figure 13b. It can be seen that there is a remarkable correlation between the average number of problems per feature and the average retrieval time. We repeated the experiment with different orders in which problems were added to the derivation history library, with similar results, further corroborating our hypothesis.

This provides a hypothesis as to when problems should be stored in the derivation history library: if adding a set of problems to the library increases the ratio problems/feature, it suggests that the new problems are quite similar to the problems already existing in the library, and hence their utility would be low. On the other hand, if the ratio decreases or remains the same, the problems are different from the ones in the library and should probably be added.

Finally, the figure shows that the retrieval time itself is not much—less than 4 seconds on average—compared to the time to synthesize programs. Again, this supports our claim about the feasibility of automatic retrieval.

6. Discussion

6.1. Issues in derivational analogy

Mostow has described a framework for evaluating replay systems (Mostow, 1989). A detailed analysis of APU in terms of this framework is given elsewhere (Bhansali, 1991). Here we focus on some of the novel aspects and limitations of APU and how they relate to other replay systems (Baxter, 1990; Blumenthal, 1990; Goldberg, 1990; Hickman & Lovett, 1991; Huhns & Acosta, 1987; Kambhampati, 1989; Mostow et al., 1989; Mostow & Fisher, 1989; Steier, 1987; Veloso and Carbonell, 1991; Wile, 1983).

Retrieval. One of the innovative features in APU is its retrieval mechanism that enables it to automatically retrieve an appropriate source analog given a description of a target problem. The retrieval heuristics are designed to estimate the closeness of two problems in the implementation domain, based on their perceived closeness in the specification domain. The surface form of problem specifications is usually not sufficient for this estimation, and one needs to either abstract certain features of the problem and reformulate them (by generalizing, canonicalizing, etc.) or compare partial solution derivations. The latter approach tends to be costly, and we have chosen the former one.

The features that are abstracted in APU are designed to represent the overall solution strategy (solution structure heuristic), the relationship between various program entities that in turn determines the structure and building blocks of the program (systematicity heuristic), syntactic cues that are associated with certain stereotyped solution structures (syntactic feature heuristic), and the similarity of the objects manipulated by the program (conceptual distance heuristic). These features are compared for various analog candidates, and the candidate that seems most closely matched is chosen as the preferred source analog.

For different domains and different specification languages, the actual manifestation of these features would be different, e.g., a different abstraction of a problem specification might determine the solution strategy. Similarly, the importance attached to the various heuristics might be different, e.g., for the domain of numerical programs, the type of a matrix (sparse, banded, symmetric, etc.) may be an important indicator of the desired solution. A fundamental limitation of the retrieval mechanism is that it is heavily dependent on abstractions for predicates, functions and objects. The next section gives certain guidelines used in forming these abstractions, but the issue needs to be explored further.

Currently, the decision of when problems and subproblems are indexed has to be made by the user. One of the important issues that needs to be resolved is to determine which subproblems to index so that the resultant increase in the retrieval time for appropriate analogs doesn't degrade APU's overall performance.

Unlike APU, some systems, e.g., POPART (Wile, 1983), BOGART (Mostow et al., 1989), XANA (Mostow & Fisher, 1989), KIDS (Goldberg, 1990), and DMS (Baxter, 1990) finesse the retrieval problem by assuming that the source analog is explicitly given by the user or that the main objective of replay is to aid design iteration. Other systems, e.g., ARGO (Huhns & Acosta, 1987), and CYPRESS-SOAR (Steier, 1987), that are based on explanation-based generalization or chunking, perform retrieval as a side-effect of rule-matching. However, as observed by Hickman and Lovett (1991), systems based on explanation-based generalization and chunking rely on complete match and direct replay of a previously learned solution, instead of partial match and adaptation, and thus have limited partial reuse capability. In ARGO, the partial reuse capability is increased by forming multiple macro-rules by abstracting a single design plan at different levels of detail.

Some of the recent work in replay does address the problem of retrieval (Hickman and Lovett, 1991; Kambhampati, 1989; Veloso and Carbonell, 1991). Unlike APU, the retrieval algorithms proposed by Kambhampati and by Veloso and Carbonell are domain-independent. Hickman's retrieval algorithm resembles APU's algorithm in using a classification hierarchy to determine when two subgoals are of the same type. However, Hickman also uses information about the success or failure of the candidate goal and the relevant information available for the source and target subgoals, before choosing the final candidate and replaying its solution.

Partial Reuse. An issue closely related to retrieval is that of partial reuse, which refers to the parts of a plan that can be replayed by themselves. The partial reuse capabilities of systems can range from one extreme where plans can only be replayed on an all-or-none basis, to the other extreme, where any subset of the design steps of a plan can be replayed. As mentioned above, EBG and chunking-based systems can replay only when a learned rule matches in its entirety, whereas systems that store the derivation trace of a problem can replay parts of a solution.

In systems like APU, which represent derivation histories as a tree, with a strict hierarchical design, subplans cannot be replayed in an arbitrary order. In particular, whenever replay fails at a particular decision node, all subsequent design decisions below the failed decision node cannot be replayed. However, in APU, subgoals of a bigger problem can be indexed independently, and since the planner always searches for new analogs when it cannot find a direct (one-step) solution to a subproblem, this limitation can be (indirectly) overcome in some cases.

An example problem that illustrates the partial reuse capability of APU is to generate a program that counts all subdirectories that are descendants of a given directory. APU first uses a source analog that counts all subdirectories (directly) under a directory, by listing the contents of the directory in separate lines, selecting those lines that have information about subdirectories, and counting the number of lines. However, when it comes to the subgoal of listing the subdirectories that are *descendants* of a directory, the analogy fails. APU then searches for and uses another analog, which lists all *files* that are descendants of a directory. This analog is a part of a larger program (the *maxfile* program discussed earlier). The rest of the parts of the original analog remain valid and APU uses it to complete the program. In this example APU uses partial fragments from two different source analogs to generate a target program, and it can be seen how it may potentially use any subset of subplans (provided they are indexed) from a bigger problem in order to solve a target problem.

A limitation of APU is that it does not use internal analogies (i.e., analogy from the current subgoal to a previously solved subgoal within the same problem) during replay. This internal analogy is possible in systems like BOGART (Mostow et al., 1989), CYPRESS-SOAR (Steier, 1987) and RFERMI (Hickman & Lovett, 1991). While in BOGART internal analogy is implemented by designating to the user the responsibility of specifying what parts to replay, in CYPRESS-SOAR and RFERMI it is implemented by dynamically accumulating the learned knowledge in the form of new productions or completely expanded goal-trees. Since APU solves subgoals in a best-first manner (using rule-levels to order subgoals), a potential analogous subgoal within the same problem would typically not be completely expanded, and could not be used to solve a new subgoal. Furthermore, if internal analogies are not very frequent in APU's domain (as represented by our experimental set), the overhead of storing and indexing the subgoals dynamically would have degraded the overall performance.

Correspondence. The correspondence problem in APU consists of determining which variables, commands, and subgoals in the new derivation correspond to which ones in a previous derivation. As in other replay systems, the correspondence is essentially achieved by unification of goals with antecedents of rules, and by assuming that terms bound to the same variable correspond. However, there are two novel features in APU. One is that it uses an AC-unification[8] algorithm (e.g., Stickel, 1981) that enables it to establish correspondence in some cases when the order of arguments of a predicate (function) is permuted. Secondly, APU uses the concept of *order* of a relation (Section 4.2) to partially canonicalize expressions. These enable it to establish the correct correspondence in cases like the following:

```
(AND (owned ?p ?u) (> (cpu-time ?p) 100)), and
(AND (≤ 10000 (size ?f)) (owned ?f ?u))
```

which would not be detected by simple unification. A detailed discussion of this and other issues related to correspondence is given elsewhere (Bhansali, 1991).

Appropriateness. One of the motivating factors in adopting a derivational analogy approach for program synthesis is that it enables one to make a decision, at each step of a plan decomposition, whether the subplan at that node should be replayed or re-synthesized. A judicious combination of replay and synthesis makes it possible to derive an efficient program efficiently.

However, as Mostow points out, there is a trade-off between the effort expended in determining the most desirable program, and the quality of the final product (in our case the efficiency of the final shell script). Thus, we need to strike a compromise, whereby if it can be easily determined that an alternative plan step is more desirable than the existing one, then the alternative step is chosen, otherwise the existing plan is replayed.

In our system, we check, before applying a replay step, whether there is a direct UNIX command or a subroutine to solve a subgoal. If there is such a command or subroutine, then it is used even though the base case had a different decomposition at that node. The resultant program in such cases is guaranteed to be more efficient modulo the assumption that the UNIX-specific rules are more efficient than other rules.

An example that illustrates this capability of APU is the synthesis of a program that replaces every occurrence of a character $?c_1$ in a file $?f_1$ with another character $?c_2$, using an analogous problem that replaces every occurrence of a word $?w_1$ in a file $?f_2$ with another word $?w_2$. The two problems are specified as follows:

```
NAME: character-substitution
INPUT: (?f1 :file ?c1 :character ?c2 :character)
OUTPUT: (?z1 :file)
PRECONDITION: true
POSTCONDITION: (= ?z1 (substitute ?c1 ?c2 ?f1))

NAME: word-substitution
INPUT: (?f2 :file ?w1 :word ?w2 :word)
OUTPUT: (?z2 :file)
PRECONDITION: true
POSTCONDITION: (= ?z2 (substitute ?w1 ?w2 ?f2))
```

Here *substitute* is a function that takes three arguments: the first two are line-objects and the third is a file. The function returns a copy of the file in which every occurrence of the first argument is replaced by the second argument. For the word-substitution problem APU produces the following partial plan:

1. **achieve** (= ?re (regular-expression ?wl))
2. sed 's/?re/?w2/' ?f2 > ?z2

This plan first computes a regular expression for the word *?wl* and then uses the UNIX command *sed* to replace every occurrence of the regular expression in the file by *?w2*. This is a partial plan because APU does not have the rules to compute the regular expression for a word. However, for the character-substitution problem, there is a single UNIX rule available that can be used to replace a character in a file with another character, resulting in the following solution:

```
cat ?f1 |
tr -s ?c1 ?c2 > ?z1
```

Since the analogy algorithm checks for a direct way of solving problems first, it finds this rule and avoids the inferior solution of the source analog.

The strategies of minimizing planning effort during replay by reusing as much of the old solution as possible, and maximizing plan quality by searching for the best alternative at each step, have been called the *satisficing approach* and the *optimizing approach*, respectively (Carbonell and Veloso, 1988). APU represents one of the first systems that considers improving the plan quality during replay, and as far as we know, there is no implemented replay system that follows a full optimizing approach.

Generality. There are several simplifying assumptions that affect the generality of the results obtained in APU. The first is concerned with the purely top-down, plan-based approach to program synthesis incorporated in APU. Baxter (1990) has pointed out the distinction between *synthesis* of a program from a specification versus *transforming* a base program given a performance predicate. Many transformation systems concentrate on the second stage of transforming a base program. In contrast systems like APU and KIDS[9] use top-down design to synthesize a base program. Because of the nature of the domain, a purely top-down approach is sufficient in APU. However, in order to be generalized, such an approach needs to be integrated with transformational rules like those used in POPART, XANA, and DMS, and their associated replay components.

Secondly, we have not investigated how derivational analogy can improve problem-solving performance by avoiding backtracking while solving analogous problems. Some results obtained in domains where the basic problem-solver has no search control knowledge seem to indicate that in such situations analogy can result in much more impressive speedups (Huhns and Acosta, 1987; Veloso and Carbonell, 1991).

Finally, the derivational analogy approach, as initially proposed by Carbonell (1983), requires a lot of detailed information to be stored in the derivation history. APU represents a simplified version of such a history. In particular, APU does not store information about failure. Some recent work suggests that information about failure can be as effective as information about success in improving analogical problem-solving performance (Hickman and Lovett, 1991).

6.2. Why does analogy work in APU?

As with most AI systems, the effectiveness of APU depends heavily on the representation of the domain. The key features of APU's representation scheme are the abstraction hierarchies of objects, predicates, and functions and the formulation of the rules in terms of these abstractions. In this section we briefly discuss how the abstraction hierarchies are formed, and what properties of the abstraction hierarchies, the rule base, and the analogical detection mechanism determine the effectiveness of APU.

Two basic guidelines in forming the abstraction hierarchy in our system are the following: (1) If a common function or predicate can be applied to objects A and B, consider classifying A and B under a more general object. For example, the operation sort-in-alphabetical-order can be applied to a set of characters, words, or lines; hence *characters, words*, and *lines* are grouped into a more general object *line-object*. (2) If a plan for achieving two goals expressed using predicates (or functions) f and g share common subgoals, consider classifying f and g into a more general predicate (function). For example, a plan for finding the largest element in an unordered collection of elements (using some ordering operator) and a plan for finding the smallest element in an unordered collection of elements share the common subgoal of first sorting the collection of elements. Therefore the predicates *largest* and *smallest* may be grouped under a common predicate called *extremum*.

One of the prerequisites for analogy to work is that there be a large proportion of general rules, i.e., rules formulated in terms of general objects, predicates and functions (hereby called *concepts*) in the abstraction hierarchy. Otherwise, if we only had specific rules written in terms of specific concepts (forming the leaves of the abstraction hierarchy), there would be very little analogical transfer of a solution derivation from one problem to another.

In addition, it seems that for analogy to succeed there should be a rich collection of intermediate (i.e., non-leaf) concepts in the domain representation. In order to see this, consider the fundamental requirement for analogy to work: *the features that are used to retrieve analogous problems should be good predictors of the sequence of rules needed to solve the problem.* Figure 14 shows the relationship between sets of problems, applicable derivations, and the features used to detect analogous problems. Different features would correspond to different sizes of the subsets of problems and the applicable derivations. A feature would be most predictive if the subset Q coincides with subset R; such a feature could be used to retrieve all, and only those, problems for which the entire derivation is applicable. However, if R is very small, such a feature would not be general enough for analogy. On the other hand, if Q coincided with the set P, then the feature would be very general, but would be a poor predictor of the subsequent rules to apply. An ideal feature for analogy is one that maximizes both the subset of problems which it identifies (for generality) as well as the part of the solution derivation that is applicable to them (for effectiveness).

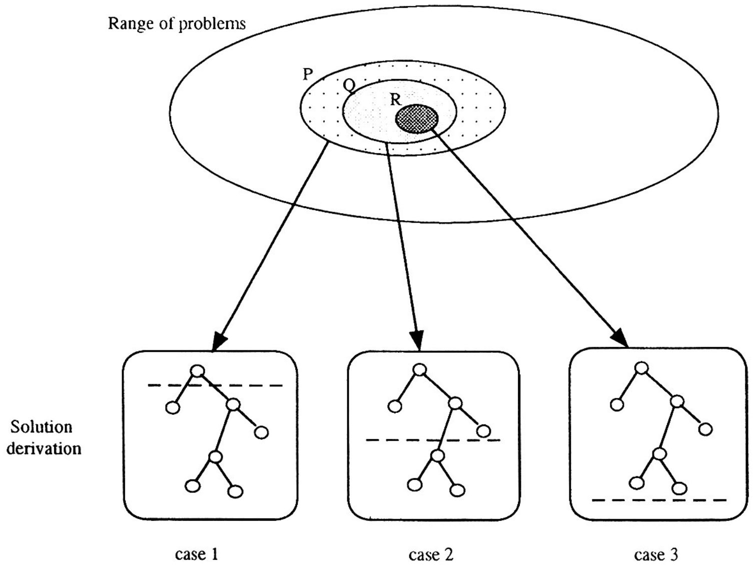


Figure 14. The relationship between problems, plans, and features used to detect analogous problems. P is the set of all problems to which the first rule in a derivation applies, Q is the subset of problems that match the feature used for plan retrieval, and R is the subset of problems for which the entire derivation is applicable.

In APU, when a rule is used to decompose a goal, the bindings of the rule variables to the goal expressions determine the subsequent subgoals and thus, implicitly, the subsequent sequence of rules to apply. Therefore the bindings of the rule variables provide a feature for analogy detection. One can imagine three ways in which these bindings can be used to predict other goals on which the same sequence of rules would apply. At one extreme, one could completely ignore the bindings of variables, and say that for any goal expression which matches the rule, the original sequence of rules should apply. However, if the rule is very general, it may be poorly correlated to the subsequent sequence of rules to be used, and thus the analogy is likely to fail as often as it succeeds. This corresponds to case 1 in figure 14. At the other extreme, one could use the exact bindings and say that if a rule matches another goal with the corresponding variables bound to the same expressions (up to variable renaming) then the original sequence of rules would apply. This corresponds to case 3 in figure 14 and is not general enough for analogy. A third, intermediate approach is to extract certain features that characterize the bindings and use them to predict the sequence of subsequent applicable rules. If the features that are used to characterize the bindings are both general (to permit analogical transfer of solutions to many other problems) and well correlated with the subsequent rules needed to solve the problem, then they can be fruitfully used to detect analogous problems (case 2 in figure 14).

The key to APU's success is that its retrieval heuristics use a set of features that provide precisely such a characterization of the variable bindings. The ontology of intermediate concepts represented in the abstraction hierarchy plays a central role in the characterization of the bindings. Because of the methodology used to construct the abstraction hierarchy, there is a strong correlation between the intermediate level concepts used to characterize variable bindings and the subsequent rule sequence that is used, and at the same time the characterization is general enough to be applicable to several different problems.

A short example should clarify the above discussion. Consider a goal expression in a a rule:

(= ?*z* (card (SET (?*x*: *object*) :ST ?*constraints*)))

where ?*z* is of type *integer*. The goal expression states that ?*z* is the cardinality of the set of objects ?*x* satisfying ?*constraints*. This expression matches a goal

(= ?*n* (card (SET (?*f* :*file*) :ST (*belongs* ?*f* ?*dir*))))

with the bindings

{?*z* = ?*n*, ?*x* = ?*f*, ?*constraints* = (*belongs* ?*f* ?*dir*)}

where ?*n* and ?*dir* are variables of types *integer* and *directory*, respectively. In English, the goal specifies that the value of ?*n* be equated to the number of files in a directory ?*dir*.

One of the features used to characterize the bindings and used during retrieval is the predicate *contained*, which is a generalization of the predicate *belongs*. This feature is general enough to select the solution of this problem to solve several other problems (counting the number of subdirectories in a directory, counting the number of words occurring in a file, etc.). At the same time the feature can be used to distinguish between plans that do not share common derivations. For example, two plans between which it can distinguish are:

(1) To determine the number of files in a directory
    list the contents of the directory,
    select lines that begin with the character "-", and
    use command *wc -l*

versus

(2) To determine the number of ancestor directories of a file
    find a plan to determine the set of ancestor directories, and
    use command *wc*

The plan for determining the ancestors of a file involves climbing up the directory structure (using the UNIX command *cd*), storing the directory name in a file, until the root directory is reached. The goals for both these problems match the rule given above, but the predicates to which *?constraint* is bound are different and do not have a common (one-step) generalization. Since this feature is used to characterize the binding, APU's retrieval mechanism is able to distinguish between the two plans.

A property of the domain that contributes to APU's success is the availability of a rich set of planning operators (i.e., the UNIX commands and subroutines) that make it possible to represent the objects in the domain in terms of abstraction hierarchies. Thus, when a plan is formulated in terms of general objects, there is a high correlation between the various plans obtained by instantiating the general object by specific ones in the abstraction hierarchy, and their completions. For example, in APU *files* and *directories* are classified under the abstract object *directory-object* and most plans are formulated in terms of *directory-object*. The analogy between files and directories is effective because for most UNIX commands that operate on *files* there is a corresponding (possibly same) command that operates on *directories*.

7. Conclusion

Derivational analogy was proposed as a powerful mechanism that could be used to reduce problem-solving effort by replaying parts of a solution trace that are applicable to an analogous problem. However, the usefulness of this approach needs to be empirically evaluated by applying this technique to non-trivial and novel domains. APU represents one of the first implemented systems that incorporates derivational analogy with automatic retrieval of base analogs in the program synthesis domain. Being a prototype system, APU has not been used to solve truly complex problems, and in fact it only addresses a simplified, though non-trivial, subset of the full program synthesis task. Nevertheless, this prototype system has demonstrated that (1) derivational analogy can be implemented using a top-down decomposition approach in a non-trivial program synthesis domain, and (2) is can speed up the program synthesis process in this domain.

We have suggested a set of heuristics for retrieving analogs for this domain and, more importantly, provided empirical evidence to show their effectiveness. These heuristics were found to be quite effective in picking a good analog for target problems, from a library of source analogs; and a retrieval algorithm based on the heuristics could be implemented efficiently enough to speed up the overall performance of the system. Although some of the heuristics and the exact form of the retrieval algorithm may not be as successful in other domains, we believe that some of the general principles on which they are based (e.g., the relationship between input and output variables in the *systematicity* heuristic, and the similarity of the objects manipulated in the *conceptual distance* heuristic) should be applicable in other domains.

Our experimental results showed that the amount of speedup (a factor of 2) obtained by using analogy was not very impressive. However, this result is not as negative as it might seem, since our experimental methodology (accepting the first partial solution) eliminated all backtracking during plan synthesis. The primary speedup was obtained by identifying parts of a solution that could be *copied* instead of replaying the sequence of rules that led to their derivation. In a more realistic setting, where the planner spends a large amount of time in backtracking, we expect the principal advantage of derivational analogy to be due to search reduction. In fact, when tried on isolated examples with backtracking turned on, APU was able to obtain speedups of as much as a factor of 12. However, we need more experimentation with a larger set of general-purpose rules in order to test the validity of this result on a population of problems.

The effectiveness of APU depends heavily on the domain representation. Some of the key features of the representation include an ontology of intermediate concepts and the formulation of rules in terms of general concepts that permit solution derivations to be applicable to several problems. For analogy to be effective, the features used to retrieve analogs should be both *general*, to permit analogical transfer of solution derivations to several problems, and *well correlated* with the sequence of rules used to solve the problems so that the plans they retrieve are likely to succeed. In APU, the ontology of concepts represented in the abstraction hierarchy plays a central role in extracting features from problem specifications with precisely such characteristics. A property of the domain that contributes to APU's success is the availability of a rich set of planning operators (UNIX commands and subroutines) that enables the creation of an ontology of useful intermediate-level concepts.

The question of the scalability of this approach is still open. An issue that needs to be resolved is that of controlling the size of the derivation history library. Our experiments suggest certain heuristics (e.g., the number of problems per feature) that can be used to decide when to store derivation histories, but this issue needs to be explored in greater depth. Another scalability issue is concerned with applying this approach to problems with significantly longer derivations. Further research is also needed to determine the generality of this approach by identifying classes of problems to which this approach can be applied. An important extension of this work is to apply it to problems that involve transformational techniques in addition to top-down planning.

Acknowledgments

We gratefully acknowledge the criticisms and suggestions of Jack Mostow and the anonymous referees, which have greatly improved the presentation of this work. This work has also benefited by suggestions from Uday Reddy and discussions with members of the knowledge-based programming group at Illinois: Kanth Miriyala, Hingyan Lee, Khaled Al-Dhaher, Sudin Bhat, Jim Ning, and Scott Renner.

Notes

1. In this case the set always consists of a singleton element since the word-count of a word is a unique integer. One may have a transformational rule that uses this information to reduce the subgoal to a simpler one, but currently APU does not have such a rule.
2. The rules in APU are not indexed, and the system finds an appropriate rule by a linear search through all rules. With an efficient indexing scheme, this information need not be stored with the derivation.
3. An independent constraint on $?x_1$ is a predicate that does not contain $?x_2$ as an argument and vice versa.
4. The order is defined as follows (Gentner, 1983): constants and variables are order 0. The order of a predicate is one plus the maximum of the order of its arguments. Thus, (size $?f$) is order 1, (> (size $?f$) N) is order 2, and so on.
5. The conceptual distance heuristic cannot be used independently, since it is not used to index problems, but simply to prune the set of candidates retrieved by the other analogs. Also, note that in the fourth experiment, with the systematicity heuristic turned off, the argument abstraction heuristic is not doing anything, since the correspondence between arguments is not established.
6. A 95% confidence interval for the mean reduction in program synthesis time is 0.51 to 0.56 for all heuristics, 0.53 to 0.6 for H_1 and 0.54 to 0.6 for H_2.
7. The Wilcoxon test is a statistical test that is designed to test the hypothesis that the mean values of two populations that have the same shape and standard deviation is the same (Mendenhall et al., 1981).
8. Associative-Commutative unification.
9. KIDS/Refine is actually a hybrid system that used both top-down design to synthesize a base program and transformational techniques to refine it into an efficient program.

Appendix

The following rules are used in the derivation of the *maxword* problem. Those variables in the rule-goal that have a UNIX command in the rule body are ordinary variables (Rules 7-13). All other variables in the rule-goal are schematic variables.

Rule R3: *To get the set of items in a collection, get multiset of the items in the collection and remove duplicates from the list.*

```
true: (= ?z (SET(?x :object) :SUCH-THAT ?conds))
    achieve (= ?s (COLLECTION(?x :object) :SUCH-THAT ?conds));
    achieve (= ?z (remove-duplicates (type-of ?x) ?s));
where
    (?z :Set(object; ?s :Collection(object))
```

Rule R4: *To get the multiset of line-objects in a stream, replace the delimiters of the text-object by a NEWLINE character.*

```
    true: (= ?z (COLLECTION(?x :line-object) :SUCH-THAT (occurs ?x ?f)))
          achieve (= ?charset (delimiters (type-of ?x)));
          achieve (= ?z (replace-chars ?charset NEWLINE ?f))
    where
          (?z :Collection(text-object); ?charset :Set(character); ?f :stream)
```

Fact 1: *The delimiters of word are (SPACE, TABS, NEWLINE)*
(= (delimiters word) {SPACE, TABS, NEWLINE})

Rule R5: *To replace a fixed set of characters by another character in a stream, replace each character in the set by the replacing character.*

```
(ground ?charset) ∧ (small-size ?charset): (= ?z (replace-chars ?charset ?char ?f))
          achieve (= $?z_1$ (replace-char cl ?char ?f));
          achieve (= $?z_2$ (replace-char c2 ?char $?z_1$));
          . . .
          achieve (= ?z (replace-char c2 ?char $?z_k$));
    where
          (?charset :Set(character); ?char :character; ?f, $?z_1$, $?z_2$,. . . :stream)
    (cl, c2 etc. are the characters belonging to ?charset)
```

Rule R6: *To find the Nth-maximum element in a set, sort the elements of the set in decreasing order and take the Nth element from the list.*

```
    true: (= ?z (Nth-maximum ?c :Nth ?n :Key ?k :Order ?rel-op)
          achieve (= ?s (sorted ?c :Key ?k :Order ?order-rel-op));
          achieve (= ?z (Nth ?n ?s))
    where
          (?z :object; ?c :Collection(object); ?s :List(object); ?n, ?k, :integer
               ?rel-op :order-op)
```

Rule R7: *To remove duplicate lines from a sorted list, use command uniq*

```
    (sorted ?f): (= ?z (remove-duplicates line ?f))
          (?z := uniq ?f)
    where
          (?z, ?f :stream)
```

Rule R8: *To remove duplicate lines from a list, sort the list and use command uniq*

```
    true: (= ?z (remove-duplicates line ?f))
          achieve (= $?z_1$ (sorted ?f));
          assert (sorted $?z_1$)
          (?z := uniq $?z_1$)
    where
          (?z, $?z_1$, ?f :stream)
```

Rule R9: *To replace character* x *by character* y *in a stream, use command* tr-s.

$$\frac{\text{(stream ?f): (= ?z (replace-char ?x ?y ?f))}}{\text{(?z := tr-s ?x ?y ?f)}}$$
where
    (?x, ?y :character; ?f, ?z :stream)

Rule R10: *To sort a list on key* ?k, *use command* sort

$$\frac{\text{true: (= ?z (sorted ?l :Key ?k :Order >))}}{\text{(?z := sort +?k -r ?l)}}$$
where
    (?l, ?z :stream; ?k :integer)

(The UNIX Sort *normally sorts in increasing order. The* -r *option is needed to reverse the order of comparison.)*

Rule R11: *To output objects to a stream use command* echo.

$$\frac{\text{(and (not (stream ?}x_1\text{)) (not (stream ?}x_2\text{)) \ldots): (output ?}x_1\text{ ?}x_2\text{ \ldots :TO ?z)}}{\text{(?z := echo ?}x_1\text{ ?}x_2\text{ \ldots)}}$$
where
    (?z :stream; x1, x2, . . . :object)

Rule R12: *To select the first element from a list, use command* head.

$$\frac{\text{true: (= ?z (Nth 1 ?f))}}{\text{(?z := head -1 ?f)}}$$
where
    (?f, ?z :stream)

Rule R13: *To select the nth field from a collection, use command* awk *if the components of the tuple are separated by whitespace (TABS or SPACES)*

$$\frac{\text{(whitespace? (field-separator ?f)): (= ?z (select-field ?nth ?c))}}{\text{(?z := awk '\{print \$?nth\}' ?c)}}$$
where
    (?c :Collection; ?nth :integer)

References

Barstow, D. (1979). *Knowledge based program construction.* New York: Elsevier North Holland.
Baxter, I.D. (1990). *Transformational maintenance by reuse of design histories.* Doctoral dissertation, Department of Computer Science, University of California, Irvine. Technical report 90-36.

Bhansali, S. (1991). *Domain-based program synthesis using planning and derivational analogy.* Doctoral dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign. Technical report UIUCDCS R-91-1701.

Bhansali, S., & Harandi, M.T. (1990a). APU: automating UNIX programming. *IEEE International Conference on Tools for Artificial Intelligence* (pp. 410–416), Washington, DC: IEEE Computer Society Press.

Bhansali, S., & Harandi, M.T. (1990b). The role of derivational analogy in reusing program design. *Fifth Annual Knowledge-Based Software Assistant Conference* (pp. 28–41). Syracuse, NY.

Blumenthal, B. (1990). Empirical comparisons of some design replay algorithms. *Eighth National Conference on Artificial Intelligence* (pp. 902–907). Boston, MA: AAAI Press/The MIT Press.

Burstein, M.H. (1986). Concept formation by incremental analogical reasoning and debugging. In R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Carbonell, J.G. (1983). Derivational analogy and its role in problem solving. *Third National Conference on Artificial Intelligence* (pp. 64–69). Washington, DC: Morgan Kaufmann.

Carbonell, J.G., & Veloso, M. (1988). Integrating derivational analogy into a general problem solving architecture. *DARPA Workshop on Case-Based Reasoning* (pp. 104–124). Clearwater Beach, FL: Morgan Kaufmann.

Dershowitz, N.D. (1985). Synthetic programming. *Artificial Intelligence, 25,* 323–373.

Dershowitz, N.D. (1986). Programming by analogy. In R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Gentner, D. (1983) Structure-mapping: a theoretical framework for analogy. *Cognitive Science, 7(2),* 155–170.

Goldberg, A. (1990). Reusing software developments. (Technical Report KES.U.90.2). Palo Alto, CA: Kestrel Institute.

Greiner, R. (1988). Learning by understanding analogies. *Artificial Intelligence, 35,* 81–125.

Harandi, M.T., & Bhansali, S. (1989). Program derivation using analogy. *Eleventh International Joint Conference on Artificial Intelligence* (pp. 389–394). Detroit, MI: Morgan Kaufmann.

Hickman, A.K., & Lovett, M.C. (1991). Partial match and search control via internal analogy. *Thirteenth Annual Conference of the Cognitive Science Society.* Chicago, IL: Lawrence Erlbaum.

Huhns, M., & Acosta, R. (1987). Argo: an analogical reasoning system for solving design problems (Technical Report AI/CAD-092-87). Austin TX: Microelectronics and Computer Technology.

Kambhampati, S. (1989). *Flexible reuse and modification in hierarchical planning.* Doctoral dissertation, Department of Computer Science, University of Maryland, College Park. Technical Report CS-TR-2334.

Kambhampati, S. (1990a). Mapping and retrieval during plan reuse: a validation structure based approach. *Eighth National Conference on Artificial Intelligence* (pp. 170–175). Boston, MA: AAAI Press/The MIT Press.

Kambhampati, S. (1990b). A theory of plan modification. *Eighth National Conference on Artificial Intelligence* (pp. 176–182). Boston, MA: AAAI Press/The MIT Press.

Katz, S., Richter, C.A., & The, K.S. (1989). PARIS: a system for reusing partially interpreted schemas. In T.J. Biggerstaff & A.J. Perlis (Eds.), *Software reusability (Vol. 1): Concepts and models.* New York: ACM Press.

Kedar-Cabelli, S.T. (1985). Purpose-directed analogy. *Seventh Annual Conference of the Cognitive Science Society* (pp. 150–159). Irvine, CA: Lawrence Erlbaum.

Mendenhall, W., Scheaffer, R.L., & Wackerley, D.D. (1981). *Mathematical statistics with applications.* Boston, MA: Duxbury Press.

Minton, S. (1988). *Learning effective search control knowledge: An explanation-based approach.* Boston, MA: Kluwer.

Miryala, K., & Harandi, M.T. (1991). Automatic derivation of formal software specifications from informal descriptions. *IEEE Transactions on Software Engineering, 17(10),* 1126–1142.

Mostow, J. (1989). Design by derivational analogy: issues in the automated replay of design plans. *Artificial Intelligence,* 40, 119–184.

Mostow, J., Barley, M., & Weinreich, T. (1989). Automated reuse of design plans. *International Journal for Artificial Intelligence and Engineering, 4(4),* 181–196.

Mostow, J., & Fisher, G. (1989). Replaying transformational derivations of heuristic search algorithms in DIOGENES. *DARPA Workshop on Case-Based Reasoning* (pp. 94–99). Pensacola Beach, FL: Morgan Kaufmann.

Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence,* 5, 115–135.

Sacerdoti, E. (1977). *A structure for plans and behavior.* Amsterdam: North-Holland.

Stefik, M. (1981). Planning and metaplanning (MOLGEN: Part 2). *Artificial Intelligence,* 16, 141–169.

Steier, D. (1987). CYPRESS-Soar: a case study in search and learning in algorithm design. *Tenth International Joint Conference on Artificial Intelligence* (pp. 327–330). Milan, Italy: Morgan Kaufmann.

Steinberg, L.I., & Mitchell, T.M. (1985). The REDESIGN system: a knowledge-based approach to VLSI CAD. *IEEE Design & Test, 2*, 45–54.

Stickel, M.E. (1981). A unification algorithm for associative-commutative functions. *Journal of the ACM, 28(3)*, 423–434.

Veloso, M., & Carbonell, J.G. (1991). Learning by analogical replay in PRODIGY: first results. *European Working Session on Learning*. Porto, Portugal: Springer-Verlag.

Waters, R.C. (1985). The programmer's apprentice: a session with KBEmacs. *IEEE Transactions on Software Engineering, 11(11)*, 1296–1320.

Wile, D.S. (1983). Program developments: formal explanations of implementations. *Communications of the ACM, 26(11)*, 902–911.

Received April 25, 1990
Accepted July 25, 1991
Final Manuscript January 9, 1992