A Model of Purpose-driven Analogy and Skill Acquisition in Programming

Peter Pirolli
University of California, Berkeley

Abstract

X is a production system model of the acquisition of programming skill. Skilled programming is modelled by the goal-driven application of production rules (productions). *Knowledge compilation* mechanisms produce new productions that summarize successful problem solving experiences. *Analogical problem solving* mechanisms use representations of example solutions to overcome problem solving impasses. The interaction of these two mechanisms yields productions that generalize over example and target problem solutions. Simulations of subjects learning to program recursive functions are presented to illustrate the operation of X.

Introduction

Theoretical progress in cognitive science hinges crucially on the ability of theories to address issues of knowledge acquisition. In turn, theories of knowledge acquisition have direct bearing on theoretical and practical issues in instruction. I present a model of analogical problem solving and skill acquisition, called X, developed as an extension of the ACT* theory (Anderson, 1983) for the domain of learning to program recursive functions. This model was developed to explore the notion insightful conceptual understandings of example solutions are instrumental in skill acquisition. Analogical problem solving makes use of goal-relevant and plan-relevant information that is encoded in a mental representation of an example solution. The content and detail of this information has an impact on the success of analogical problem solving. In turn, skill acquisition mechanisms transform these analogical problem solving experiences into skills. Ultimately, the quality of the skills acquired from analogy rests upon the content and detail of the goal-relevant and plan-relevant information in the example representation. Analogy thus serves as a means towards effecting *repairs* (Brown & Van Lehn, 1980) to domain-specific problem solving procedures.

Analogy is a term with many denotations and connotations in cognitive science as well as everday language. The particular analogical problem solving mechanisms implemented in X work from representations of example solutions that might be presented by a textbook or teacher. Basically, these representations constitute an explanation of the structure and functionality of components of the solution structure along with constraints on the conditions under which such components apply. In this respect, the X analogical problem solving mechanisms are similar to machine learning work on explanation-based learning (DeJong & Mooney, 1986; Mitchell, Kellar, & Kedar-Cabelli, 1986). The invocation of analogical problem solving and the selection of relevant analogs is driven by active problem solving goals. This aspect of X is similar to machine learning work on purpose-driven analogy (Kedar-Cabelli, 1985). Although similar in spirit to these clusters of machine learning research, X evolved from the ACT* production system theory of problem solving and is a minor variant of another descendent of ACT*, the PUPS production system (Anderson & Thompson, 1986). Like other members of the ACT* family, accounting for phenomena of human cognition is one major impetus for developing X. Another motivation for developing X, is the notion that richer theories of learning will lead to richer, more effective, and more efficient means of instruction.

Programming Recursion

An Ideal Model

The X model was initially developed to address observations of people learning to program recursive functions. Recursion is usually a novel concept for introductory programming students and consequently serves as a useful domain for studying knowledge acquisition. A typical example of a recursive function in LISP is the definition of the function FACT to compute factorials presented in Table 1. Like all recursive functions, FACT is defined in terms of itself. The body of the definition for FACT consists of a conditional structure containing a series of conditional clauses. The conditional structure is implemented by the LISP form COND and each conditional clause is represented as a list within the COND structure (a list in LISP is anything enclosed in parentheses; e.g., (A B C) is a list). Each conditional clause contains two parts. The first part is a list that specifies a condition and the second part specifies an action to take if the condition does not evaluate to NIL, which stands for "false." The first clause in FACT states that if the input N is equal to zero, then the result of FACT will be 1. The second clause, states that in all other cases, the result of FACT will be N times the result of FACT applied to N - 1.

Table 1

A recursive LISP function to compute factorials

Definition	Comments
(DEFUN FACT (N)	; define n!
(COND ((ZEROP N) 1)	;If n = 0 then return 1
(T (TIMES N (FACT (SUB1 N))))))	;Else return n x (n - 1)!

In previous studies of expert and novice programming (Pirolli & Anderson, 1985) we identified concepts that appeared to be crucial to efficiently learning general skills for programming recursion. A prescriptive model of programming skills for recursion, called an *ideal model*, has been implemented in the GRAPES production system and used as the basis for instruction on recursion in an intelligent tutoring system (Pirolli, in press). GRAPES is a system that emulates a subset of the ACT* theory (Anderson, 1983).

Recursive functions, in general, have a conditional structure consisting of two types of cases. The *recursive cases* compute a result by using the results of one or more *recursive calls* to the function. For example, the last conditional clause of the FACT function presented in Table 1 is a recursive case that involves computing the result of the recursive call, (FACT (SUB1 N)), and uses that result to compute the result of (FACT N). The *terminating cases* terminate the recursive process. In Table 1 the terminating case is the first conditional clause, which returns the value 1 when the input N = 0.

The *recursive relation* holds between the value of the function and the value of a recursive call to the function. In FACT, the recursive relation is the relation between between the result of $n!$ and $(n - 1)!$--that is, $n! = n \times (n - 1)!$. Students often have a great deal of difficulty identifying and planning out the recursive cases of recursive functions. Based on our analyses of expert problem solving with recursive functions (Pirolli & Anderson, 1985) the general method for determining the recursive relation is to: (a) assume that the result of a recursive call can be achieved, then (b) determine how to achieve the result of the function using the result of the recursive call.

GRAPES productions, in general, decompose programming goals into subgoals until some action can be achieved, forming a hierarchical goal tree. A typical production for programming LISP might be:

P1: IF the goal is to write a function definition in LISP
and the name of the function is given
and the arguments to the function are given
THEN write "(DEFUN <name> <arguments> <body>)"
where <name> is the name of the function
and <arguments> are the arguments to the function
and set a subgoal to code the <body> of the function
which implements a process that achieves the function

Some observations on analogical problem solving

Instruction in problem solving domains usually includes descriptions of the entities in the domain, general rules for problem solving, and example solutions. Protocol studies (Pirolli & Anderson, 1985) have indicated that subjects have a tough time deriving solution methods from general definitions or rules of thumb for a domain and turn to example solutions for guidance in early problem solving attempts . These subjects are usually successful in producing some solution by analogy to an example once the example has been selected (see also Gick & Hoyoak, 1980; Reed, Dempster, and Ettinger, 1985; Reed, Ernst, and Banerji, 1974). We also observed (Pirolli & Anderson, 1985) that problem solving procedures are learned from such analogical problem solving (see also Gick & Holyoak, 1983; Sweller & Cooper, 1985). However, subjects sometimes conceptualize examples in a shallow and literal manner and sometimes conceptualize examples in a deep and insightful manner (Pirolli & Anderson, 1985). These variations have an impact on the problem solving behavior generated by analogy to examples. In some cases subjects basically copy program code that they do not understand and in other cases they produce solutions based on the more principled methods they see embodied in the example programs. These variations in early problem solving experiences involving the use of examples in turn have an impact on early skill development (Pirolli & Anderson, 1985).

The pervasiveness of analogical problem solving in the early stages of skill acquisition is borne out by an experiment in which subjects had access to an online example of a recursive function, that had been part of their instruction. While coding their first recursive function, subjects spent approximately 30% of their time looking at the example. Analysis of target problem solutions revealed that portions that were similar to the example accounted for fewer errors. Later, when subjects were coding their fourth recursive function, looking at the example solution only accounted for about 5% of their problem solving time.

The impact on skill acquisition of the particular manner in which an example is encoded is illustrated by another experiment in which the same example program was presented to one group of subjects (structure group) in the context of an explanation of how recursive functions are written (based on the GRAPES ideal model) and to another group (process group) in the

context of how recursive functions execute (the typical pedagogical approach). The structure group took less training time than the process group in achieving the same level of proficiency indicating that knowing how an example recursive function is written yields more efficient learning than knowing how a recursive function works. A simulation of a structure group subject is presented later in this paper.

The X Model

The X model of analogical problem solving was implemented as an extension of the GRAPES production system. The X model is also a subset of a production system architecture for analogical problem solving and skill acquisition, called PUPS, that is being developed by Anderson and Thompson (1986). Basically, X takes several ideas developed in PUPS about analogical problem solving and instantiates them in the GRAPES architecture.

Like several other proposals for problem solving by analogy (e.g., Carbonell, 1986; Gick & Holyoak, 1980, 1983) the X analogy mechanisms supply a method for problem solving when domain-specific methods are lacking or inadequate. The general notion is that the learner has some declarative knowledge of how the structure, S_e, of an example achieves various functions, F_e, under certain preconditions, C_e, and is faced with achieving goals G_t ,under conditions C_t in a target problem. The task in analogical problem solving is to come up with a target solution S_t by solving the analogy $S_e:F_e::S_t:G_t$, subject to the constraint that the mapping of S_e onto S_t transforms C_e, into a set of preconditions that are in C_t or satisfiable in the target solution.

Representation

The X system makes use of a representation scheme for declarative knowledge that captures the functionality, structure, and conditionality, of concepts or actions in a problem solving domain. This knowledge representation scheme is crucial to the working of analogy and is an important addition to the ACT* theory. The principle components of this scheme are schematic knowledge structures called *units* that have slots that are filled or instantiated by particular values. Although arbitrary slots are allowed, there are three types of slots that have preeminance in the representation: (a) *functionality* which describes the purpose or goals achieved by a unit, (b) *structure* which describes the composition of a unit from other units, and (c) *conditionality* which describe constraints on the unit.

Table 2

Examples of X representations

The POWER program.

power-definition
functionality: defines(power-function args power-result)
preconditions: implemented-in(power-function LISP)
structure: steps(defun power-name args body)

The Factorial Problem.

fact-definition
functionality: defines(fact-function x-arg fact-result)
preconditions: implemented-in(fact-function LISP)

Some examples of the representation can be seen in Table 2, which presents a declarative description of part of an example program and a target problem in LISP. The example is a recursive definition of a function, POWER, that computes m^n. The target program is the factorial function presented in Table 1. Units thus provide a knowledge representation scheme that captures important goal-relevant and plan-relevant information for use in problem solving. A unit can be thought of as rule of the form

$$conditionality \wedge structure \Rightarrow functionality$$

So, *power-definition* in Table 2 can be translated into the rule

$$steps(defun\ power\text{-}name\ args\ body) \wedge implemented\text{-}in(power\text{-}function\ LISP) \Rightarrow defines(power\text{-}function\ args\ power\text{-}result)$$

Problem solving

Goals in the X system are to-be-achieved units that have functionality but no structure. An example of a to-be-achieved goal in the Table 2 is *fact-definition*. The X system considers one goal at a time and considers only productions that are applicable to that goal. The propositions on the structure slots represent orderings, partial orderings, and hierarchical relationships among the actions represented by units. The agenda for goal processing is achieved by productions that search through the structural links from a current active goal. Units encountered in this search that have a specified functionality but no structure are placed on the goal agenda. Analogical problem solving is invoked by X at problem solving impasses--in other

words, when a goal is activated and no production matches, analogical problem solving is invoked. X selects an analog for further processing based on a specificity principle. In theory, this is an associative memory retrieval achieved by the spreading activation mechanisms of ACT* (Anderson, 1983; Anderson & Pirolli, 1985). In the computer implementation of X, the effects of spreading activation are approximated by a specificity principle based on the number of correspondences and identical elements that hold between the the example and target. Since the goal of analogical problem solving is to map an existing solution structure from an analog unit to a target unit, one constraint on the selection of an analog unit is that it must have a filled-in structure slot. There are three major subprocesses involved in solving a target problem by analogy:

- *Function matching.* the first step taken by X is to place the target goal unit into correspondence with the functionality of potential analogs and to select the best analog. Two function propositions can be placed into correspondence if the predicates in both functions are identical. The arguments of one function proposition are placed into correspondence with a another function proposition by virtue of the slots they fill within the propositions. These correspondences are used to map information about the analog onto new information in the target. Function matching also checks that the conditions on the analog unit are not violated in the target problem. If there is a violation then there is no match.

- *Structure mapping.* This involves mapping an analog structure onto a new target structure. However, there is no guarantee that the correspondence set will be elaborate enough to permit such a mapping.

- *Function elaboration.* This occurs when an element of an analog's structure has no correspondence. There are a number of ways that function elaboration can be carried out in order to map a particular structural element e_a of an analog onto a new structural element e_t in a target. First, the functionality of e_a may match the functionality of some existing target unit e_t. The correspondence set can be elaborated with this correspondence plus the correspondences resulting from the function match of e_a and e_t. Second, a new target unit can be created and assigned a functionality mapped from e_a and this may recursively invoke further function elaboration. Third, additional correspondences can be found by elaborating the match of an analog unit to a target unit. This is achieved by recursively matching the functions of elements already placed into correspondence. This may lead to an elaborated set of correspondences that permits e_a to be mapped onto a new e_t.

Learning from Analogy

One major outcome of analogical problem solving is the induction of new production rules by a set of *knowledge compilation* mechanisms that generalize over information present in the declarative units representing analog and target problems and their solutions. Knowledge compilation mechanisms create new productions that summarize the problem solving involved in analogy (for the details of knowledge compilation see Anderson, 1983). These new productions apply in situations similar to those that invoked analogical problem solving in the first place. The compilation of solutions produced by analogy yield general problem solving operators and thus the interaction of analogy and knowledge compilation offers an alternative procedure for the generalization of cognitive skills in ACT* (Anderson, 1986).

To see how skills are acquired from analogy, consider that analogical problem solving in X consists of two things: (a) matching conditions and functionalities of the analog and target, and (b) creating target conditions, functionalities, and structures based on the analog. Knowledge compilation creates new productions with conditions that specify the target functions and conditions that were matched in the analogy process and that have actions that specify the structures, functions, and conditions that were created in the target by analogy. The conditions created by compilation retain the components of the target that matched exactly to the analog and variablizes over target information that mismatched. Thus, a compilation of the analogical problem solving involved in achieving the goal unit *fact-definition* based on the example *power-definition* produces the production:

```
L1:  IF  the goal is to achieve
         =definition
              functionality: defines(=function =arguments =result)
              precondition:  implemented-in(=function LISP)
         and
         =name
              functionality: name-of(=function)
     THEN the structure of =definition is
              steps(defun =name =arguments =body)
         and the functionality of =body is
              implements(=function)
```

where the items preceded by the equal sign denote variables. Production L1 applies when the goal is to achieve a function definition in LISP when the name of the function has been decided on. The action specified by L1 lays out a template for the code to define the function. Production L1 has the same semantics as production P1 presented earlier.

Example Simulations

Previous protocol analyses and experiments on learning recursion (Pirolli, 1985; Pirolli & Anderson, 1985) indicate that subjects with richer representations of how recursive functions are written learn more efficiently and effectively than subjects who either just understand how recursive functions operate or who have a encoded a rather literal representation of examples. With X it is possible to explore in greater detail what knowledge promotes efficient and effective learning. Two simulations of X are presented here. These simulations address verbal protocol data analysed and modelled previously (Pirolli & Anderson, 1985) in a more general manner. The first simulation illustrates how a rather literal representation of an example can lead to a successful solution by analogical problem solving, with very little gain in skill acquisition. The second simulation illustrates a case of analogical problem solving that lead to effective problem solving skills for programming recursion.

Literal Analogy

The first simulation addressed the data gathered from subject JP, an eight year old learning recursion in LOGO. Her first programming problem was to write a function that would recursively draw a set of squares of increasing size. JP's final solution, called TUNNLE [sic] was

```
TO TUNNLE :X
SQUARE :X
IF :X = 42 THEN STOP
TUNNLE :X +10
END
```

In coding TUNNLE, JP used an example program, CIRCLES, to guide her analogical problem solving. After writing TUNNLE (which works), JP was unable to code even slight variants of TUNNLE (e.g., drawing more squares).

The analysis of protocol data and interview data suggested that JP has a very literal representation of the CIRCLES program and largely copied that solution onto the TUNNLE [sic] solution. Figure 1 presents the representation of CIRCLES that is encoded initially in X for the simulation of JP. In Figure 1, literal code from the example is in uppercase. The structure of the CIRCLES code is not represented at any deeper level (e.g., as a tree structure representing the different LOGO structures, terminating cases, recursive cases, etc.). The functionality of only some of the program symbols are elaborated (e.g., CIRCLES is the name of the function, "50" is the maximum size of the circles).

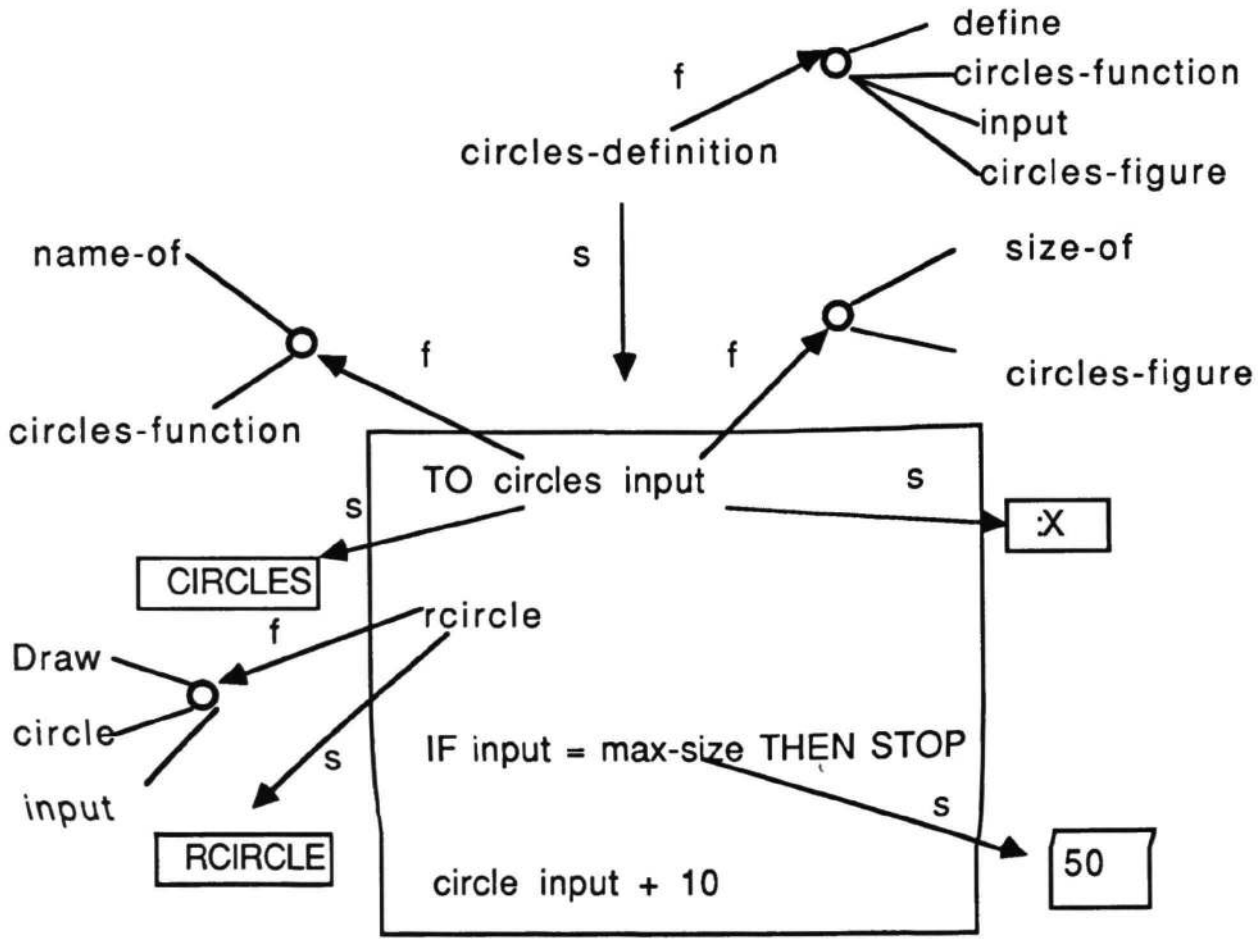

Figure 1: JP's representation of the CIRCLES example. Arrows labelled *s* indicate structure slots; *f* indicates function slots. Boxes indicate structures that fill structure slots.

Analogical problem solving works in this case because the target problem is similar enough to the example representation to permit successful mappings. The following matches are made in the simulation:

- Both definitions define a function taking an input and produce a composite figure
- The process implemented by both functions is to repeatedly draw a figure of increasing size
- RCIRCLE draws an element of the composite circles figure and SQUARE draws an element of the composite squares figure
- 50 is the maximum size of the circles figure and 42 is the maximum size of the square figure
- The functionality of the inputs match

Knowledge compilation of this analogical problem solving yields the following production

```
L2:  IF  the goal is to define a function =name with input =x
         that draws a =composite-figure
         by repeatedly drawing a figure with a function =figure-drawer
         up to maximum size =number
     THEN write
              TO =name =x
              =figure-drawer
              IF =x = =number THEN STOP
              =name =x + 10
```

Production L2 will basically code other fuctions that draw composite figures of increasing size like CIRCLES, but is not effective for coding recursive functions in general.

Insightful Analogy

The second simulation addressed data gathered from subject AD, a college student learning recursion in SIMPLE (Shrager & Pirolli, 1983). AD's programming tasks centered on writing functions that searched and gathered selections from a database of library entries. These library entries could be identified by a number or title, and SIMPLE predicates were available to test whether entries belonged in one of three categories (science, religion, or fiction). The recursion problems assigned to AD involved collecting library entries of various categories into lists with different orderings placed on the list items. AD's instruction on recursion was identical to that given to the structure group in the experiment discussed earlier--that is, it emphasized how recursive functions are written. The example discussed in this instruction was SORT, a function that sorted a list of entries such that science books were at the beginning of the list result of SORT.

From AD's protocol gathered as she read instructions out loud and wrote her first recursive function, it was clear that she had encoded a rich representation of the SORT example. After writing her first recursive function, AD coded an additional 19 recursive functions without error. Part of the encoding of SORT given to X in simulating AD is presented in Figure 2, which depicts the representation of a recursive case of SORT. The representation includes the notion that the recursive case is a conditional structure and that the action in this case involves a recursive relation. The recursive relation is achieved by determining the result of a recursive call to SORT and then comparing it to the result of SORT.

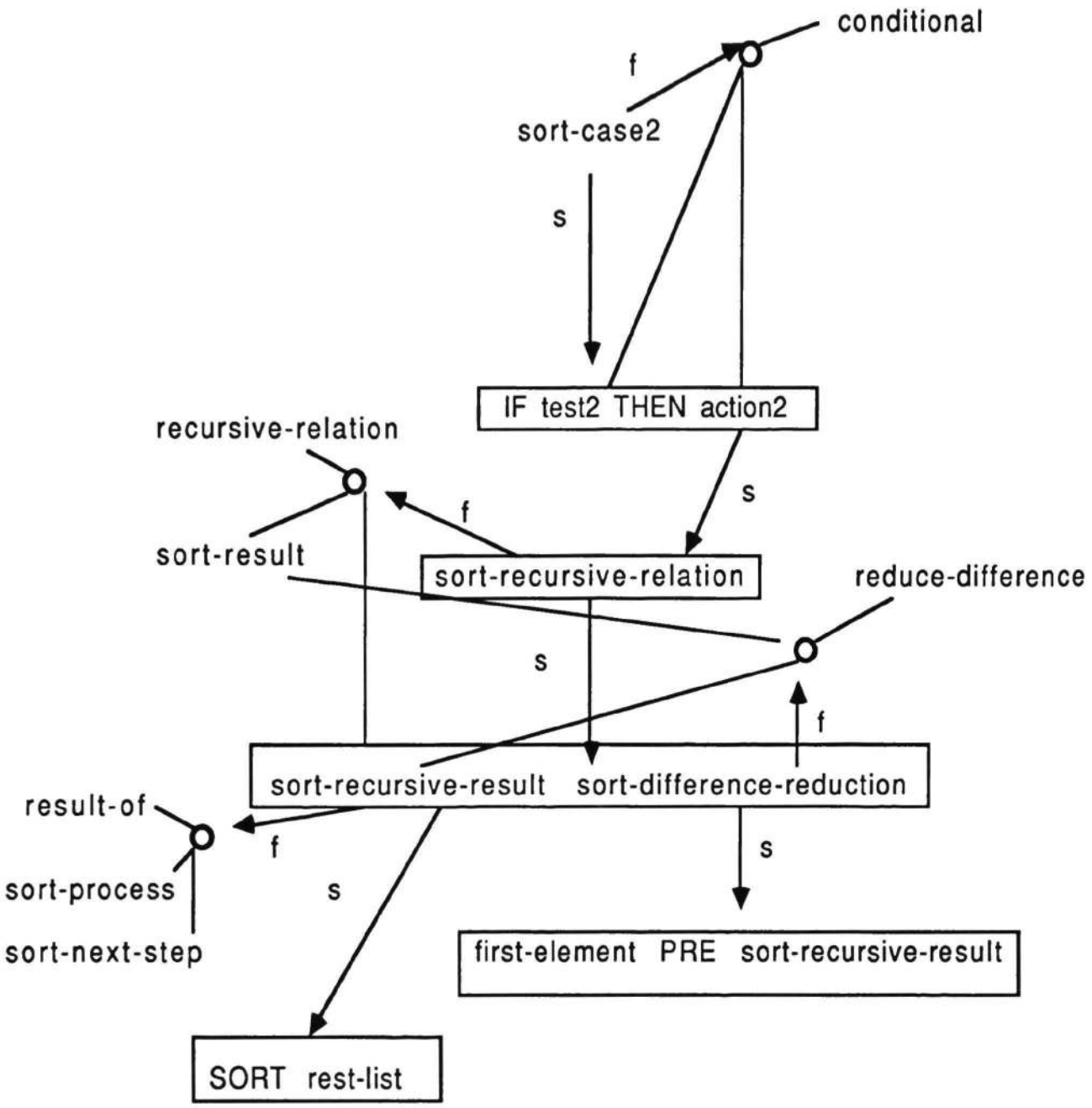

Figure 2: AD's representation of the recursive case of the SORT example. Arrows labelled *s* indicate structure slots; *f* indicates function slots. Boxes indicate structures that fill structure slots.

Knowledge compilation of AD's analogical problem solving yield the following productions

L3: IF the goal is to code the action of a conditional clause
of a function that places all elements
of a specified type into a result in a specified order
THEN set a goal to code the recursive relation

L4: IF the goal is to code a recursive relation
THEN set subgoals to
1. refine the recursive call
2. achieve the result of the function using the recursive call

L5: IF the goal is to code a recursive call to a function
and a value that is one step closer
to the terminating case than the current function input is known
THEN write the name of the function followed by that value

L6: IF the goal is to achieve a value that is one step closer
to the terminating case than the current function input
and the function input is a number
THEN set a goal to write code that will subtract 1 from the input

Production L3 specifies that the action of a conditional clause that is supposed to place all elements of a certain type in an output list can be solved by coding the recursive relation. Production L4 lays out a plan for inducing a recursive relation to satisfy the constraints of the program. Production L5 codes a recursive call. Production L6 specifies that the argument to a recursive call in a function that has a numeric input should be one less than the input. The key point to be made in this simulation of subject AD is that having an abstract representation of the underlying structure and functionality of a recursion example that encoded how recursive functions are written facilitated the learning of productions that are similar to those of the ideal model implemented in GRAPES. The generality of the productions acquired by X in simulating subject AD accounts for the ease with which AD coded her subsequent recursion problems.

Summary

The X model of analogical problem solving and skill acquisition was developed as an extension of ACT* (Anderson, 1983) to deal with the pervasive phenomena of analogical problem solving. A major difficulty with ACT* has been with its ability to deal with the structuring of problem solving performance when encountering a novel domain (e.g., Anderson, 1983, ch. 6). The analogical problem solving mechanisms in X (and PUPS, Anderson & Thompson, 1986) comprise a weak problem solving method that appears to serve this function in a number of domains such as programming, geometry (Anderson, Greeno, Kline, & Neeves, 1981), and algebra (Reed et al., 1985; Sweller & Cooper, 1985). The interaction of analogy and knowledge compilation yields generalized productions from a single problem solving episode. This appears to more accurately fit the phenomena of early skill acquisition (e.g., Kieras & Bovair, 1986) better than the ACT* mechanism of production generalization which requires two similar production applications.

The major gap in X is that it does not address the process of comprehending example solutions to form the representation that serves as the basis for later analogical problem solving. Current work is focused on filling this gap. It is assumed that understanding an example is driven by a process of attempting to explain (by instantiating declarative schematic knowledge about plans and goals) how an example solution achieves various goals (cf. DeJong & Mooney, 1986). The goal of this effort is to work towards a model that addresses some aspects of how variations in prior knoweledge about plans and goals interact with variations in instruction using examples.

References

Anderson, J.R. (1983). *The architecture of cognition.* Cambridge, MA: Harvard University Press.

Anderson, J.R. (1986). Knowledge compilation: The general learning mechanism. In R.S. Mickalski, J.G. Carbonell, and T.M. Mitchell (Eds.) *Machine Learning, Volume 2.* (pp. 289-310). Los Altos, CA: Morgan Kaufmann.

Anderson, J.R., Greeno, J.G., Kline, P.J., and Neves, D.M. (1981). Acquisition of problem-solving skill. In J.R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 191-230). Hillsdale, NJ: Lawrence Erlbaum.

Anderson, J.R. and Pirolli, P.L. (1984). Spread of activation. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 10*, 791-798.

Anderson, J.R. and Thompson, R. (1986). *Use of analogy in a production system architecture.* Unpublished manuscript, Carnegie-Mellon University, Department of Psychology, Pittsburgh, PA.

Brown, J.S. and VanLehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science, 4*, 379-426.

Carbonell, J.G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R.S. Mickalski, J.G. Carbonell, and T.M. Mitchell (Eds.) *Machine Learning, Volume 2.* (pp. 371-392). Los Altos, CA: Morgan Kaufmann.

DeJong, G. and Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning, 1*, 145-176.

Gick, M. L. and Holyoak, K. J. (1980). Analogical problem solving. *Cognitive Psychology, 12*, 306-355.

Gick, M. L. and Holyoak, K. J. (1983). Schema induction and analogical transfer. *Cognitive Psychology, 15*, 1-38.

Kedar-Cabelli, S. (1985). *Purpose-directed analogy.* In Proceedings of the Cognitive Science Society. Irvine, CA: CSS.

Kieras, D.E. and Bovair, S. (1986). The acquisition of procedures from text: A production-system analysis of transfer of training. *Journal of Memory and Language, 25*, 507-524.

Mitchell, T.M., Kellar, R.M., and Kedar-Cabelli, S.T. (1986). Explanation-based generalization: A unifying view. *Machine Learning, 1*, 47-80.

Pirolli, P.L. (1985). *Problem solving by analogy and skill acquisition in the domain of programming.* Unpublished doctoral dissertation, Carnegie-Mellon University.

Pirolli, P. (in press). A cognitive model and computer tutor for programming recursion. *Human-Computer Interaction.*

Reed, S.K., Dempster, A., and Ettinger, M. (1985). Usefulness of analogous solutions for solving algebra word problems. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 11*, 106-125.

Reed, S.K., Ernst, G.W., and Banerji, R. (1974). The role of analogy in transfer between similar problem states. *Cognitive Psychology, 6*, 436-450.

Pirolli, P.L., and Anderson, J.R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology, 39*, 240-272.

Shrager, J. & Pirolli, P. L. (1983). *SIMPLE: A simple language for research in programmer psychology* [Computer program]. Pittsburgh: Carnegie-Mellon University, Department of Psychology.

Sweller, J, and Cooper, G.A. (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction, 2*, 59-89.