Assessing Transfer of a Complex Skill

Irvin R. Katz

Division of Cognitive and Instructional Science
Educational Testing Service
Princeton, NJ 08541

katz@clarity.princeton.edu

Abstract

While recent studies have demonstrated various ways that transfer might be achieved in a domain, the measures used to assess transfer rarely stray from time and error data. This paper examines transfer in the complex skill of computer programming in order to explore more flexible and sensitive methods of assessing transfer. In the experiment, subjects wrote both a PASCAL and a LISP version of two programming problems. Although a simple accuracy measure provides evidence for knowledge transfer between the two programming languages, measures based on analyses of the task domain (i.e., partial-credit accuracy, strategy use) provide much stronger evidence. Curiously, these measures target different subjects as exhibiting transfer, suggesting that more than one type of knowledge may be available for transfer.

Introduction[1]

Transfer is the phenomenon in which knowledge or skills learned in one context affect the learning or performance of another task. Not too long ago, an article on transfer would start with an argument for why transfer might be found between two domains and end with inconclusive results (Gray & Orasanu, 1987). However, the recent literature has seen a wave of studies demonstrating the various ways that transfer can be achieved through careful analysis of the task domains (e.g., Singley & Anderson, 1990).

While the methods used to achieve successful results have progressed, there has been little change in the ways that transfer is measured. In a typical experiment, subjects receive training in one domain and then are asked to perform tasks in another domain. Evidence for transfer is reported when subjects perform the transfer task either faster or more accurately than subjects without the benefit of training. One reason for this use of time and error data might be because transfer has been investigated in fairly simple, structured domains (e.g., Tower of Hanoi: Smith, 1986; text editing: Polson, Muncher, & Engelbeck, 1986). Those studies that use other measures of transfer, such as subjects' use of a particular problem-solving strategy, tend to involve more complex domains like physics and algebra (e.g., Bassok & Holyoak, 1989).

Unlike most studies of transfer, the motivating question for this research is not "Does transfer occur?" but rather "How should transfer be assessed?". This paper describes an experiment investigating transfer of computer programming knowledge. By using computer programming as the task domain, we can explore a greater variety of transfer measures than that possible in more structured domains. Computer programming has the added advantage that although it is a complex, semantically-rich domain, it nevertheless has been intensively analyzed by many researchers (Pennington, 1985). These prior analyses form a basis for understanding how transfer might occur in the computer programming domain.

Because the motivation for this work is an interest in the nature of any knowledge transfer that might occur, the ideal experimental task would be one in which the subjects are likely to try and use previously generated knowledge. As discussed by Gick & Holyoak (1987), when the surface characteristics of two tasks are very similar, it is likely that subjects will attempt to use information learned from one task while solving the second task. Thus, in the experiment, subjects were asked to write two versions of the same programming problems, the versions differing only in the required implementation language, PASCAL or LISP.

Most models of programming skill would predict that transfer should occur in this situation. When writing a program, typically a programmer devises some initial mental plan of how the program should work, then begins implementing that plan. Through the process of coding the program, difficulties with the plan may be uncovered,

[1]This research was supported in part by a National Science Foundation Graduate Fellowship and by Contract MDA903-85-K-0343 from the Army Research Institute and the Air Force Human Resources Laboratory. Preparation of this manuscript was supported by a grant from the Japan Society for the Promotion of Science. I thank John Anderson, Jerry Feigenbaum, and Claudius Kessler for their comments on various portions of this work.

requiring the plan to be refined or changed (Green, 1987). The final result of this processing is not only a program, but also the refined plan for writing the program, the latter of which forms the programmer's mental representation of the just-written program This representation is often characterized as containing abstract knowledge independent of the particular computer language used to implement the program (Rist, 1989).

The programmer's representation serves as the starting point for the implementation of the second version of the program in the new language. The coding of the second version proceeds as if this transferred information were the initial plan developed specifically for the second version. As a result, in coding the second version, transfer should occur because the program's plan need not be generated from scratch. This notion of transfer of planning knowledge is supported by verbal protocol data (Katz, 1988). Subjects make less verbalizations referring to the program plan while writing the second version of a program, but produce an equal number of verbal statements referring to the actual program code for both versions of a program.

Method

Subjects. Subjects were 21 students who had just completed a four week LISP mini-course. All subjects had already known PASCAL before entering the LISP course.

Materials. Subjects were given two problems, called Printnums and Addfract, and wrote a LISP and PASCAL version of each problem. The Printnums problem was to write a program that translates a given number between 1 and 999 into its corresponding words. Thus, if given "115," the program should output "ONE HUNDRED FIFTEEN." Subjects were provided with functions that returned the appropriate units-, teens, or tens-word given a digit (e.g., the tens-word function returned "FIFTY" when given "5"). The Addfract problem was to write a program that accepts two fractions in numerator-denominator form and returns the reduced sum of the fractions, also in numerator-denominator form. Thus, given "5 / 6" and "7 / 15" the program should return "13 / 10." Preliminary analyses revealed that these two problems are of approximately equal difficulty.

Design. As stated above, all subjects wrote both LISP and PASCAL versions of the two problems — four programs total. The ordering of the actual problems received was counter-balanced. Thus, when discussing the results, the particular serial order of programs will not be considered.[2]

[2]An additional manipulation — whether or not subjects' previous programs were available for reference — was included in this experiment, but did not affect any of the reported measures.

Procedure. After writing two simple warm-up programs (one each in PASCAL and LISP), subjects were given the first problem description. The programs were written using a standard screen editor, and subjects were able to test their programs whenever they wished. If a subject did not write a working program within 30 minutes, the subject was stopped and then given the instructions for the next problem. Thus, at the most, the experiment lasted approximately 2.5 hours, including the warm-up exercises.

Results and Discussion

While subjects overall had trouble writing perfectly working programs (approximately 65% of the programs written did not execute perfectly), by ignoring syntax errors and mis-understandings of the problem descriptions, there is evidence for knowledge transfer between programming languages (Table 1). For the first version of the programs, only 48% of the programs were correct. In contrast, 71% of the second versions were correctly written, and this interaction is significant by a chi-squared test ($\chi^2(1) = 4.94$, $p<.05$). Clearly, subjects gained knowledge while writing the first version of their programs that they could apply in writing the second version.

Considering the closeness of the two tasks, however, it is strange that so little transfer occurred. Only about one-quarter of the subjects showed the expected accuracy improvement. Overall accuracy might not be a particular sensitive measure of transfer. We need more flexible assessments of subject performance, ones that are based on an analysis of the problem-solving tasks.

	Correct	Incorrect
First Version	20	22
Second Version	30	12

Table 1: Program accuracy

Partial Credit. Using a partial credit scheme for scoring accuracy provides a more sensitive measure of transfer. Thirty minutes turned out not to be enough time for subjects to write a completely working program. Thus, some of the programs written by subjects were incomplete (i.e., missing some essential component(s) of a working algorithm). However, if subjects reused their knowledge in writing the second version of a program, then they should have been able to progress further in writing the second version than they did in writing the first. In other words, they should have been able to "pick up where they left off" writing the first version.

In order to judge the completeness of the programs, a general algorithm was enumerated for each of the two problems, Printnums and Addfract. As an example, the Printnums algorithm is shown in Table 2. While there are

alternative algorithms for the problems, the other algorithms include approximately the same pieces as the ones enumerated, except in a different order. The parts of the algorithm that the subject tried to achieve were recorded for each subject's program. It was not necessary for a particular algorithm piece to be completely correct — the important point is that the subject tried to achieve that piece. This scoring of each program pair (first and second versions) was done by a judge blind as to which of the two programs in a pair was actually written first.

1. Get number
2. Is there a hundreds digit?
3. Calculate hundreds digit
4. Call units function with above value
5. Print/include result of (4)
6. Print/include "hundred"
7. Remove hundreds digit
8. Is there a separate tens digit?
9. Calculate tens digit
10. Call tens function with above value
11. Print/include result of (10)
12. Remove tens digit
13. Handle separator ("-")
14. Is the number a teen?
15. Call teens function on number
16. Print/include result of (15)
17. Is there a units digit?
18. Call units function on number
19. Print/include result of (18)

Table 2: Printnums algorithm

If subjects reused information, then those subjects who wrote an incomplete program for the first version should have written a second version including all the pieces of the first version, plus some more. In contrast, a lack of transfer would show as the two versions being incomplete in different ways. For example, the two versions might consist of different patterns of missing and implemented algorithm pieces, or the second version may be more incomplete than the first. Finally, if the two versions are both complete, it is ambiguous as to whether or not transfer has occurred.

As shown in Table 3, in 77% of the unambiguous cases (i.e., excluding the "Both Fully Complete" column), subjects' second version was more complete than the first version, even though the two versions were written in different computer languages. This result is a strong indication of knowledge transfer; subjects used information learned in the writing of the first version when writing their second version of the program in a new language. By re-using this knowledge, subjects built on work done previously to produce a more complete program.

Pgm. Name	More Complete	Less Complete	Both Fully Complete
Printnums	11	3	7
Addfract	12	4	5

Table 3: Partial credit analysis

Strategy Use. Another method for assessing transfer is to observe subjects' use of problem-solving strategies. In the simplest case, possible strategies would include one correct method for solving the problem and several incorrect methods (e.g., "balance" vs. "switch" strategies in the Missionaries-Cannibals problem and its isomorphs; McDaniel & Schlager, 1990). In a complex domain such as programming, there may be a number of strategies that all lead to a correct solution. Each of these strategies would result in programs with different overall organizations. For investigating transfer of strategy use, the problem Printnums was analyzed because subjects solved this problem in two very different ways.

The first algorithm used to solved this problem is to categorize the given number into one of a set of mutually exclusive categories. For example, some subjects would categorize the number as either a one, two, or three digit number and then do the appropriate actions. The important feature of this solution is that, at the top-level, control is given to only one set of actions. Subjects usually implemented this algorithm using a single, nested conditional statement, so this algorithm will be referred to as the "nested" condition structure.

The second solution involves breaking the number into its component digits and performing appropriate actions for each of the hundreds, tens (or teens), and units digits. Control-flow might proceed where first a decision is made about the hundreds digit, and different actions are performed depending on whether or not that digit is zero (i.e., if a hundreds digit exists). In either case, however, control comes back together to handle the remaining two digits of the number. The distinguishing characteristic of this algorithm is the recombination of control. Subjects usually implemented this algorithm using multiple, separate conditional statements, so this algorithm will be referred to as the "separate" condition structure.

For the analysis, each of the 21 pairs of programs written by subjects was categorized as reflecting either a nested or a separate algorithm, or as ambiguous. As described above, the categorization of a program as nested or separate is objective: if the program has a single, top-level conditional statement, it is categorized as nested. If at the top-level of the program there are more than one conditional statements, the program is categorized as separate. Ambiguous programs were so incomplete that a distinction between the algorithms could not be made.

Two subjects wrote one ambiguous program apiece, and the pairs of programs from these subjects were eliminated from all further analyses, resulting in 19 pairs of programs.

The argument for transfer in this case goes as follows. First one must demonstrate subjects' preferences for the two algorithms under normal circumstances. That is, without having written another program previously, which algorithm are subjects more likely to use? When writing the second version, transfer would be evident if, as a result of writing the first version, subjects show a different pattern of biases toward the two algorithms.

For the first versions of each program pair, there was an effect of language on algorithm choice. As shown in Table 4, subjects programming in PASCAL showed a strong tendency to use a separate condition structure while the LISP subjects showed no such preference. Although the PASCAL subjects' bias is significantly different from chance ($\chi^2(1)=6.4$, $p<.025$), the interaction is only marginally significant ($\chi^2(1)=2.9$, $p<.10$).

	Nested	Separate
PASCAL	1	9
LISP	4	5

Table 4: Condition structure choices in each language (First Version)

As for performance on the second program, subjects clearly preferred to use the same structure that they had used on their previous program, even though that program was written in a different language (Table 5). If no transfer had occurred, the previously mentioned language-induced biases would have been evident on the second program as well. Instead, 14 subjects kept the same general algorithm on both programs, while only 5 subjects switched to the other algorithm type ($\chi^2(1)=4.26$, $p<.05$).

	Second Version	
First Version	Nested	Separate
Nested	4	1
Separate	4	10

Table 5: Condition structure choices for each version

So far, we have seen evidence for significant levels of transfer along two fronts. Subjects' programs were more complete on the second versions and the subjects tended to use the same methods for implementing each version of their programs. The measures based on analyses of the programming problems, the partial credit and strategy use transfer measures, provide much stronger and unambiguous evidence of transfer as compared with a simple accuracy measure.

Clearly subjects are generating knowledge while writing their first versions that they are able to use in writing the second versions of their programs. Thus, there should be a perfect correlation between the two measures of transfer. Subjects who re-use information should show transfer on both measures while other subjects should show a lack of transfer no matter how transfer is assessed.

As shown in Table 6, this simple model of transfer is simply wrong; there is no evidence for inter-dependence between the two measures of transfer. Approximately half of the subjects who demonstrated accuracy transfer in the partial credit scheme did not demonstrate strategy transfer — even though the subjects wrote more complete programs for their second versions, those programs did not always reflect the same algorithm as was used on the first version. The same is true for subjects who did not demonstrate accuracy transfer. Finally, subjects who wrote fully complete programs for both versions, and thus did not provide evidence either for or against transfer, used the same algorithm each time.

	Algorithm used on First vs Second Version	
	Same	Different
More Complete	6	4
Less Complete	1	1
Both Fully Complete	7	0

Table 6: Partial credit and strategy transfer

The two transfer measures do not correlate probably because they are measuring different kinds of transfer. When writing a program in one language, subjects use knowledge previously generated while writing the same program in another language, but that knowledge may consist of more than one type of information. This is an important result because most transfer studies attempt to assess only one type of transferred information for a given problem situation. What are the two sorts of knowledge that are indicated here?

During programming, a programmer breaks the main goal of the program into a set of less complex, more manageable subgoals. Through this activity, a programmer can focus on implementing just certain subgoals, which is an easier task than trying to keep in mind all of the requirements of the programming problem (Fisher, 1986). Furthermore, these subgoals must be combined so as to

represent how the program works. For example, Kant & Newell (1984) argue that the subgoals are organized in a way representing the flow of data through the program. Thus, the programmer's representation should consist of at least two kinds of information: programming subgoals and the way that those subgoals are organized. McDaniel & Schlager (1990) note a similar distinction in their study of a Missionaries-Cannibals isomorph. Solving that puzzle involves knowing both the correct general strategy and the particular moves needed to implement the steps (subgoals) of the strategy.

These two types of program knowledge are separately assessed by the partial credit and strategy use transfer measures. Through the partial-credit measure, we assess whether or not subjects transfer information about the specific subgoals used in the first version of the problem. The strategy measure shows whether or not subjects transfer information about how subgoals are organized. Thus, the assessment methods based on analyses of the task domain demonstrated that more than one type of transfer may occur in a single problem-solving situation.

References

Bassok, M., & Holyoak, K. J. 1989. Interdomain transfer between isomorphic topics in algebra and physics. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 15(1):153-166.

Ericsson, K. A., & Simon, H. A. 1984. *Protocol Analysis: Verbal Reports as Data.* Cambridge, MA: MIT Press.

Fisher, C. 1986. How do programmers program: Coping with complexity. Unpublished manuscript, Department of Psychology, Carnegie Mellon University.

Gick, M. L., & Holyoak, K. 1987. The cognitive basis of knowledge transfer. In S. M. Cormier and J. D. Hagman eds., *Transfer of Learning: Contemporary Research and Applications.* New York, NY: Academic Press.

Gray, W. D., & Orasanu, J. M. 1987. Transfer of cognitive skill. In S. M. Cormier and J. D. Hagman eds., *Transfer of Learning: Contemporary Research and Applications.* New York, NY: Academic Press.

Green, T. R. G. 1987. Parsing and gnisrap: A model of device use. In G. M. Olson, S. Sheppard, and E. Soloway eds., *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Kant, E., & Newell, A. 1984. Problem solving techniques for the design of algorithms. *Information Processing & Management* 20(1-2):97-118.

Katz, I. R. 1988. Transfer of knowledge in programming. Ph.D. dissertation, Department of Psychology, Carnegie Mellon University.

McDaniel, M. A., & Schlager, M. S. 1990. Discover learning and transfer of problem-solving skills. *Cognition and Instruction* 7(2):129-159.

Pennington, N. 1985. Cognitive components of expertise in computer programming: A review of the literature. *Psychological Documents* 15(2702).

Polson, P.G., Muncher, E., & Engelbeck, G. 1986. A test of the common elements theory of transfer. In Proceedings of the CHI '86 Conference on Human Factors in Computing Systems, 78-83. New York, NY: ACM.

Rist, R. 1989. Schema creation in programming. *Cognitive Science* 13:389-414.

Singley, M. K., & Anderson, J. R. 1990. *The Transfer of Cognitive Skill.* Cambridge, MA: Harvard University Press.

Smith, S. B. 1986. Transfer of learning between Tower of Hanoi isomorphs. Ph.D. dissertation, Department of Psychology, Carnegie Mellon University.