Skill Acquisition Can Be Regarded as Program Synthesis: An Integrative Approach to Learning by Doing and Learning by Analogy

Ute Schmid (SCHMID@CS.TU-BERLIN.DE)
Fritz Wysotzki (WYSOTZKI@CS.TU-BERLIN.DE)
Institute of Applied Computer Science, Technische Universität Berlin
Franklinstr. 28/29, 10587 Berlin, Germany

May 19, 1998

Abstract

In this paper we propose an approach to skill acquisition which is based on a technique for inductive program synthesis developed in the domain of automatic programming. This approach enables us to model skill acquisition as generalization on three levels: In a first step, learning by doing is performed by generalizing over problem states which were explored when solving a given problem. This process is similar to compilation or chunking of production rules. But in contrast to these approaches, we represent procedural knowledge as conditional programs. In a second step, descriptive generalization of the initial conditional program is performed. A recursive program scheme is constructed which generalizes over recursive enumerable problem spaces. In a third step, learning by analogy is performed by abstracting from the concrete semantic of the operation symbols contained in a recursive program scheme. The abstract scheme represents the class of structurally identical problems. By describing, how problem schemes can be constructed as generalization over knowledge gained during solving concrete problems, our approach gives an unifying framework for describing learning by doing and learning by analogy. Additionally, we consider the acquisition of some types of motor and process control behavior as a special variant of the acquisition of problem solving skills, and demonstrate, how acquisition of behavioral skills can be integrated in our framework.

1 Introduction

In Artificial Intelligence as well as in Cognitive Psychology, acquisition of problem solving skills is usually described as construction of (production) rules. Disposal of such rules enables a (cognitive) system, to solve succeeding problems "automatically", without the need to perform complex search and inference processes (Anderson, 1983). An usual claim is that problem solving skills are acquired during problem solving processes, that is by **learning by doing** (Anderson, Conrad, & Corbett, 1989). Anderson (1983) describes the automatisms on which problem solving is based as "procedural knowledge" (know how) in contrast to "declarative knowledge" (know that). In Cognitive Science procedural knowledge is usually represented by production rules. In approaches to problem solving, where not the problem solving skills but (declarative) knowledge about problem structures is the prominent aspect, usually problem schemes or frames are used as representation format (Novick & Holyoak, 1991; Anderson & Thompson, 1989).

Problem solving is generally modelled as heuristic search in a problem space (Newell & Simon, 1972). There are standard procedures for forward search (as A*, Nilsson (1980)) as well as goal driven production systems (ACT-R, Anderson (1993); SOAR, Newell (1991)). In both frameworks the definition of problem operators together with conditions for their application - i.e. production rules - is central. Selection and application of production rules is performed by an interpreter: the interpreter selects and applies a production rule to data in working memory until the problem solving goal is reached or until no rule is applicable. Production rules can be regarded as conditioned programms without loops. All "knowledge" about cyclic processing of operations is contained in the interpreter. An usual claim in Cognitive Science is, that the strategy of the interpreter is part of a fixed general cognitive architecture and that only the knowledge represented in the production rules can be subject of changes due to skill acquisition (Klahr, Langley, & Neches, 1987)). This restriction of learning to chunking of production rules makes it difficult to model learning of processing strategies as for example the cyclic structure of operator application to solve the Tower of Hanoi problem (Shell & Carbonell, 1989). But there is empirical evidence, that people can acquire such kind of knowledge (Klix & Rautenstrauch-Goede, 1967; Anzai & Simon, 1979).

An alternative approach to the acquisition of problem solving skills by learning by doing is **learning by analogy**. A new problem is solved by analogical reasoning (Gentner, 1983), that is, the solution of a structural similar example problem is adapted to solve the new problem[1]. Learning is usually described as generalization over the common structure of example and goal problem (Novick & Holyoak, 1991; Anderson & Thompson, 1989). Anderson and Thompson (1989) demonstrates, how analogical reasoning and learning can be incorporated into a production system architecture (see also ACT-R, Anderson (1993)). Example problems and solutions are represented as "scheme-like structures". New problems are represented in the same way. Empty slots of a current problem (i.e. the new solution) are filled by structure mapping between example and new problem. Learning is done by compilation of the trace of the analogical process, thereby gaining production rules containing the information formerly represented in the problem schemes. An alternative approach is, as mentioned above, to model analogical learning by the construction of generalized problem schemes.

Anderson's proposal does not address the following aspects: (1) The schemes for representing the structure of problems and solutions are represented in the system from the beginning. Thereby Anderson supposes, that the (cognitive) system has already knowledge about the relevant structure of the problem domain. But the crucial deficit of novices is that they may have *no* knowledge about the relevant problem structures Novick (1988), Schmid and Kaup (1995). (2) There is no statement about the kind of problem structures which can be handled by the proposed approach to analogical mapping and transfer. The examples in Anderson and Thompson (1989) are restricted to generalized

[1] so called "within domain analogies" in contrast to the "between domain analogies" as for example considered by Gentner (1983), see Vosniadou and Ortony (1989)

problem isomorphs, i.e. identical structures where variable and operation symbols can be substituted. Empirical studies demonstrate, that people also can use non isomorphic examples to reason and learn by analogy (Pirolli & Anderson, 1985; Reed & Bolstad, 1991). Both aspects are generally seldom treated in cognitive science: There are very few proposals for modelling the acquisition of schemes (Rumelhart & Norman, 1978, 1981); see also (Cornuejlos, 1997). While there are some empirical studies using non isomorphic examples in analogical reasoning (see above) there is no systematic classification of types of structural similarities which can be used in analogical reasoning and learning. This is also true for most computational models on analogical problem solving[2] (Falkenhainer, Forbus, & Gentner, 1989; Holyoak & Thagard, 1989).

We propose a computational model for learning by doing and learning by analogy which covers all aspects dicussed above. That is, our model describes:

- building behavioral programs from problem solving experience as an alternative approach to compiliation/chunking of production rules,
- induction of problem schemes by generalization over behavioral programs, thereby giving an formal approach to model acquisition of schemes which represent knowledge about cyclic (recursive) structures of operator applications in problem solving,
- analogical problem solving and learning on the basis of program schemes, introducing first ideas how to deal with non isomorphic structures.

We do not propose, that our approach is another unifying cognitive architecture (Anderson, 1993; Newell, 1991). We do not claim, that the knowledge representation formats and inference techniques employed in our model are adequate descriptions of human cognitive processes. But we try to model certain aspects of human problem solving for the existence of which there is some empirical evidence in an uniform and formally sound way.

In the following sections we first give an overview of our computational model. Then we present its different aspects in some detail. That is, we describe how behavioral programs can be constructed from problem solving experience. Than we introduce a method of inductive synthesis of recursive program schemes and demonstrate the application of such schemes in analogical problem solving and learning. In the last section we show, that our approach can also be applied to model some types of the acquisition of motor and process control behavior.

2 An integrative framework for skill acquisition

To give the general idea of our approach we present an example of a simple blocks world problem (see figure 1). There are three blocks A, B and C stapled to a tower. The problem solving goal is to unstack the tower, that is to clear the bottom block C. The system has one operator - $puttable(x)$ - and its application condition (x must be free to be put on the table; $cleartop(x)$) at disposal. The problem space consists of three states: the three-block tower, the tower where A is already on the table and the goal state (C is free and A and B are on the table). When decomposing the three-block-tower

[2] A lot of work is done on the influence of different degrees of surface similarity, that is semantic or pragmatic features of example and goal problem (Novick, 1988; Holyoak & Thagard, 1989).

for the first time, the system has to check whether there is an operator at hand which is applicable to the initial state. Application of the operator transforms the state into a new state. If this state is not the goal state, the system has again to select an operator which is applicable and so on. If the system is confronted with a problem for the first time, it has to be solved by heuristic search. Problem solving gives the system experience what operators can be applied to which state to transform it into the goal state. This experience can be represented as a conditional program with predicates characterizing states and corresponding operator sequences, where constants are replaced by variables (a more comprehensive description is given in the following section). This first kind of generalization over problem states is an alternative way to model skill acquisition by knowledge compilation or chunking in production systems (Rosenbloom & Newell, 1986; Anderson, 1986). If the system is confronted with the same or another initial state of an already solved problem, the constructed conditional program can be used to solve the problem in an effective way, omitting heuristic search.

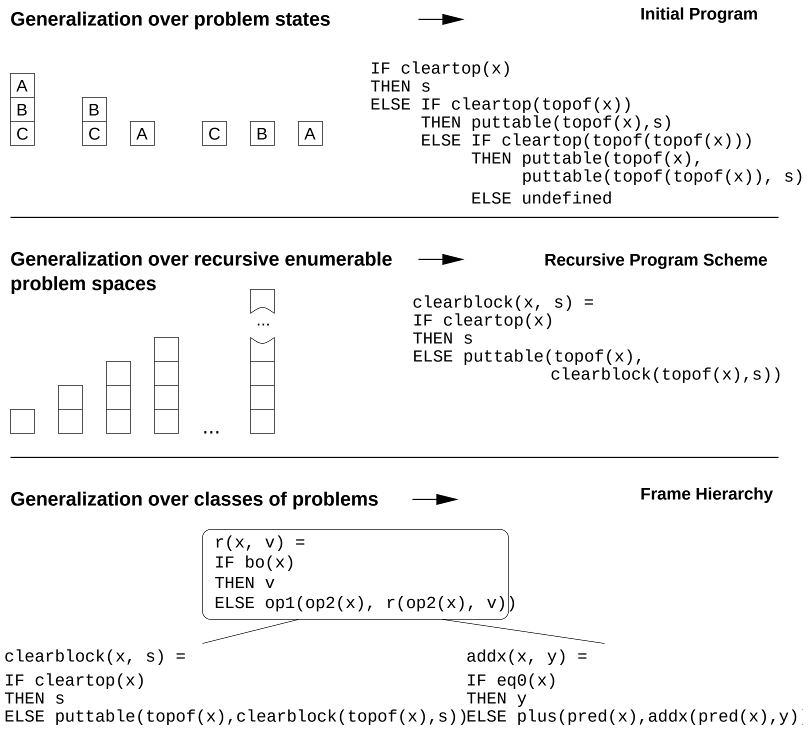

Figure 1: Skill acquisition as three levels of generalization

We believe, that experience in solving a special problem not only results in the acquisition of automatisms for solving this problem, but that additionally a generalized knowledge structure is inferred. If a problem solver has gained experience in decomposing towers in a three-blocks world, this experience helps him also to decompose towers in an n-blocks world. That is, the problem solver generalizes over recursive enumerable problem spaces. This kind of descriptive generalization can be modelled by a method of inductive synthesis of recursive program schemes (RPSs; see Wysotzki (1983), ? (?) and section 4). This representation format corresponds to problem schemes (or frames) and has the characteristics, that it simultanuously catches the *structure* of a problem (considered as a scheme) and that it is executable (as a program after appropirate interpretation). The notion of schemes as programs was also employed by (Rumelhart & Norman, 1981).

RPSs can be considered mathematically to be elements of a term algebra, that is they are not restricted to a special programming language. Operation symbols contained in a RPS can be interpreted semantically by different operations. Thereby, a RPS represents not only a structure of how to solve a specific problem but a structure for solving a class of structural isomorphic problems. This characteristics can be exploited for analogical transfer. If a behavioral program (RPS) with a certain recursive (cyclic) structure exists, a structural isomorphic (or structural similar) problem can be solved by analogical transfer omitting the effort of inducing an RPS from scratch (see section 5).

Our approach is incorporated into the architecture of the experimental system IPAL which is implemented as a first prototype in LISP (see figure 2). Input to the system is a set of problem states, a goal state and a set of operators and their conditions for application. For each problem state the optimal sequence of operators to transform it into the goal is constructed by an problem solving algorithm. Combining these sequences to an initial program realizes the first step of generalization. In the next step, it is checked, whether there is a sufficiently similar problem in the knowledge base. An RPS can be expanded to an initial program being able to handle a prescribed finite number of cases only. If there exists an RPS in the knowledge base whose initial program has a structural similarity higher than S to the current initial program, the new RPS is constructed by analogical transfer. Otherwise the new RPS is inferred by descriptive generalization with a inductive program synthesis technique mentioned above. The new RPS is stored in memory. If the new RPS was gained by analogical transfer, additionally a generalized RPS is constructed which becomes the parent of the both RPSs involved in the analogical process. Generalized schemes can be used in the same way as the example RPSs for (analogical or rather abstraction based) reasoning. Thereby the knowledge structere may become hierachically organized with learning experience.

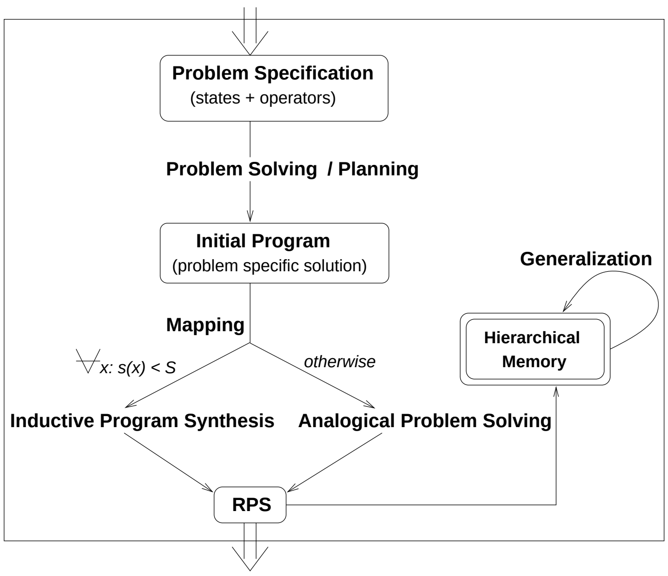

Figure 2: System Architecture of IPAL

3 Acquisition of problem solving skills

Now we describe the first step of generalization, i.e. the construction of initial programs from problem solving experience, in more detail. For illustration we use the "clearblock"-Problem (see figure 1). The three problem states are represented by conjunctions of predicates:

$$\begin{array}{ll} (1) & cleartop(A),\ cleartop(B),\ cleartop(C) \\ (2) & on(B,C), cleartop(A), cleartop(B) \\ (3) & on(A,B), on(B,C), cleartop(A). \end{array}$$

The goal predicate is $cleartop(C)$. The operator puttable is represented as production rule:

$$on(x,y),\ cleartop(x) \rightarrow puttable(x),$$

with the semantics ADD *cleartop(y)* DEL *on(x,y)*.

Additionaly we use the selector function $topof(x) = y \Leftrightarrow on(y,x)$.

The problem solver tries to construct a solution for each initial state. State (1) fulfills the goal already, no operator has to be applied. State (2) fulfills the condition part of the production rule. The rule is instantiated with $\{x/B, y/C\}$ and $puttable(B)$ is applied. Operator application transforms state (2) into state (1), the goal is fulfilled. In state (3) the operations $puttable(B)$ and $puttable(C)$ are executed.

The construction of an initial program is preformed by following steps:

- Constants are represented constructively, in dependence from the constants occuring in the goal predicate. In the clearblock problem we rewrite $B = topof(C)$ and $A = topof(topof(C))$.
- We use situation calculus Green (1969) to represent operator sequences as terms. That is, we add an additional parameter s (a situation variable) to each operator. The operator $puttable(x)$ in the clearblock problem is rewritten to $puttable(x,s)$.
- The remaining constants are variabilized[3]. Thereby a generalization over the given problem space (figure 1) is achieved. If C is replaced by a variable x, we regard all situations where an fixed but arbitraty block x is clear as goal states. This block x has not to be placed on the table but can be on top of other blocks.
- A minimal conjunction of predicates is calculated, which discriminates all problem states associated with different operator sequences (see table 1). This is a typical concept acquisition problem and can be handled with decision tree techniques given in the machine learning literature. We use the decision tree algorithm CAL2 proposed by Unger and Wysotzki (1981). In the clearblock example, we have to discriminate three states which can be achieved by the single predicate *clearblock*. In the goal state $clearblock(x)$ is true, in the second state $clearblock(topof(x))$ is true and in the third state $clearblock(topof(topof(x)))$ is true.

[3]Variablization can be considered as a by-product of learning: If production rules are applicable to constants C and expressions $topof(C)$, information about the data structure of the problem domain can be extrapolated.

Table 1: Problem states and associated operator sequences for the clearblock problem

(1)	cleartop(x)	s
(2)	cleartop(topof(x))	puttable(topof(x),s)
(3)	cleartop(topof(topof(x)))	puttable(topof(x),puttable(topof(topof(x)),s))

- The predicates are ordered by complexity of terms (which corresponds to a complete partial order over the data type) and are composed to a conditional expression. The resulting initial program for the clearblock problem is given in figure 1.

We are exploring an alternative approach for constructing initial programs which (1) enables us to deal with problems with higher complexity, especially with goal states which are composed of dependent subgoals[4]; and (2) which provides a different representation format for initial programs which makes inductive program synthesis more simple for some problems (see section 4). Instead of the two step procedure of heuristic search and decision tree construction, the initial program is built in one step by a goal-driven planning algorithm (see Wysotzki (1987), Schmid and Wysotzki (1996)). The resulting plan is a binary tree with the goal as root node. Left branches represent cases where the top predicates are true, right branches apply the operators to achieve the top predicates. That is, the plan corresponds to a conditional program. We demonstrate this planning approach for the clearblock problem.

Operators are represented by basic programs instead of production rules. For each predicate regarded in the goal state a corresponding basic program is given which describes what to do, if the predicate is true or if the predicate is not true in a situation:

$$G_{cleartop(x)} = cleartop(x)(s|\ |, puttable(topof(x))(s|cleartop(topof(x))|)).$$

The right hand side of the basic program is to be read as "if cleartop(x) is true than situation s has not to be changed, else apply the operator $puttable(topof(x))$ but first ensure that $cleartop(topof(x))$ holds". The operators given in basic programs are annotated at the right branches of the plan. The conditions for operator application given in the basic programs are introduced as new goals. The initial program for the clearblock problem constructed by planning is given in figure 3.

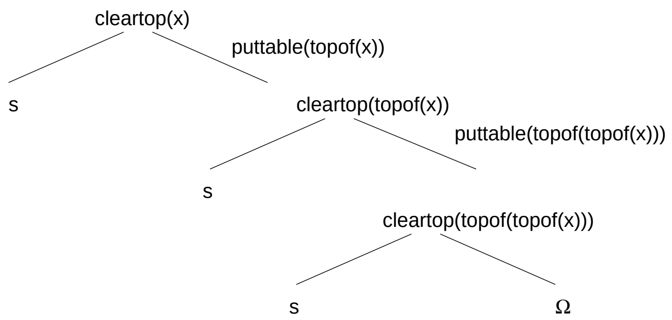

Figure 3: Initial program constructed by planning (Ω means "undefined")

[4]For example to reach the goal $on(A, B)on(B, C)$ the subgoal $on(B, C)$ has to be achieved first.

4 Scheme induction

To model descriptive generalization over recursive enumerable problem spaces we use a method of inductive program synthesis introduced by Wysotzki (1983) and further developed and implemented by ? (?). Alternative approaches to this problem were presented in the context of planning (iterative macro-operators: Shell and Carbonell (1989)) and in explanation-based learning (n-generalization: Shavlik (1990)).

To grasp the general idea of the method, consider numerical induction problems (see also Gold (1967)) as used in intelligence tests, for example $1, 4, 9, 16, \ldots$. To give the next element of this sequence, one have to infer the rule, by which the element at position i ($i = 1$ to 4) can be constructed. As more elements are taken into consideration as more the set of possible rules gets restricted. If you look only at the first two elements, for example the rule $e^{(i+1)} = e^{(i)} + 3$ and the rule $e^{(i)} = i \times i$ are covering the data. But the first rule do not hold for the next two elements. Of course, the second rule, which covers all four elements of the sequence is also only hypothetical. If the next element would not be 25 a new rule have to be considered. But for the examples at hand, the observed regularity is best covered by our second rule. In the following, we employ the idea of discovering regularities and perform an inductive generalization Holland, Holyoak, Nisbett, and Thagard (1986) not for simple numerical expressions, but for terms. We demonstrate the method with the clearblock problem. The theoretical background and a formal description of the approach is given in ? (?).

First we take the initial program given in figure 3 as input to inductive program synthesis. We use the McCarthy conditional $g(x, y, z) =_{Def}\ IF\ x\ THEN\ y\ ELSE\ z$ to represent it as term with prefix notation only:

$$
\begin{aligned}
\mathcal{G} \;=\;& g(cleartop(x), s, puttable(topof(x), \\
& g(cleartop(topof(x)), s, puttable(topof(topof(x)), \\
& g(cleartop(topof(topof(x))), s, \Omega))))).
\end{aligned}
$$

The initial program represents the problem solving experience gained in a three block world when working on the clearblock problem. The symbol Ω (undefined) indicates, that the system has gained no experience for the case that the second block on a block x is not clear. Descriptive generalization will now be performend by extrapolation over the structure of the initial program. The general idea is, to identify a term tr corresponding to a subtree starting at the root in $\mathcal{G}$ which occures repeatedly with a subsitution Θ of variables v by terms t. To this aim we decompose $\mathcal{G}$ in a sequence of terms $G^{(i)}$ with $G^{(i+1)} = tr(G^{(i)}_{\Theta})$. In terms of (theoretical) computer science that is, we interprete $\mathcal{G}$ as an element of a Kleene sequence (linear expansion) belonging to some recursive program. By definition $\mathcal{G}^{(0)} = \Omega$ holds. Now we can rewrite $\mathcal{G}$ as sequence

$$
\begin{aligned}
G^{(0)} &= \Omega \\
G^{(1)} &= g(cleartop(x), s, puttable(topof(x), \Omega)) \\
G^{(2)} &= g(cleartop(x), s, puttable(topof(x), \\
&\quad\; g(cleartop(topof(x)), s, puttable(topof(topof(x)), \Omega) \\
G^{(3)} &= g(cleartop(x), s, puttable(topof(x),
\end{aligned}
$$

$$g(cleartop(topof(x)), s, puttable(topof(topof(x)),\\ g(cleartop(topof(topof(x))), s, \Omega))))) = \mathcal{G}$$

with $tr = g(cleartop(x), s, puttable(topof(x), m))$ and $\Theta = \{x/topof(x)\}$. Symbol m marks the place where a subterm can be inserted into the structure. We can rewrite the sequence by

$$\begin{aligned} G^{(0)} &= \Omega \\ G^{(1)} &= g(cleartop(x), s, puttable(topof(x), G^{(0)}_{\Theta})) \\ G^{(2)} &= g(cleartop(x), s, puttable(topof(x), G^{(1)}_{\Theta})) \\ G^{(3)} &= g(cleartop(x), s, puttable(topof(x), G^{(3)}_{\Theta})) \end{aligned}$$

and extrapolate

$$G^{(i+1)} = g(cleartop(x), s, puttable(topof(x), G^{(i)}_{\Theta}))$$

for all $i \in \mathcal{N}$, thereby getting in the limit the recursive program scheme

$$G(x, s) = g(cleartop(x), s, puttable(topof(x), G(topof(x), s))).$$

Taking the inital program given in figure 1 as input to program synthesis, the procedure gets more complicated (see figure 4a for a representation as decision tree). The problem is, that each THEN-branch represents a complex operator sequence which have to be decomposed for extrapolating an recursive program scheme. This can be done by rewriting the operator sequences introducing new variables for counting operator applications and dealing with the THEN-branches as seperate induction problem:

$$\begin{aligned} H^{(0)} &= s \\ H^{(1)} &= puttable(topof'(1, x)) \\ H^{(2)} &= puttable(topof'(1, x), puttable(topof'(2, x), s)) \end{aligned}$$

can be extrapolated to

$$H(i, n, x, s) = g(equal(i, n), s, puttable(topof'(i, x), H(succ(i), n, x, s)))$$

where i is a counting variable for the number of topof-applications and n is the number of puttable-operations which have to be performed. The clearblock-RPS is:

$$\begin{aligned} G(x, s) &= G'(0, x, s) \\ G'(n, x, s) &= g(cleartop(topof(n, x)), H(0, n, x, s), G'(succ(n), x, s)). \end{aligned}$$

$G'(n, x, s)$ only calculates the number of blocks n lying on to of x. This number is input to the additional function H which handles the application of the *puttable* operations.

While descriptive generalization of the clearblock problem is simple when using plans as initial programs, it is complicated and results in a more complex solution for initial programs gained by forward application of production rules. The problem arises in all cases where the problem structure is not a simple tail-recursion[5]. In figure 4b an initial program for a simple tail-recursive problem is given. The problem solving goal is to put the uppermost block of a tower on the table. The initial program can be folded to the RPS $T(x) = g(cleartop(x), puttable(x), T(topof(x)))$.

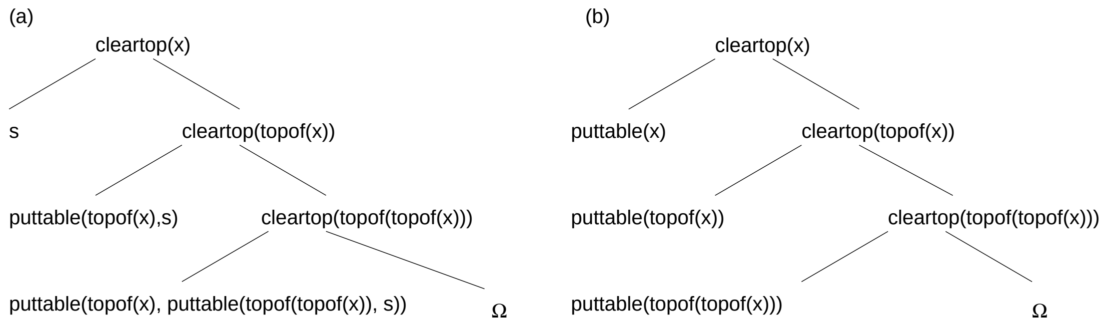

Figure 4: Initial programs constructed from forward application of production rules (Ω means "undefined")

That the representation of a problem has effects on problem solving success is well known from psychological studies (c.f. Simon (1970)). Our approach suggests additionally, that the problem solving strategy used by a (cognitive) system influences the ease of generalization.

5 Analogical problem solving

Analogical problem solving can be described by four processes (see Novick and Holyoak (1991)):

- *Retrieval*: Search for an example problem already solved which is structural similar to the current (goal) problem.
- *Mapping*: Compare the structures of example and goal problem. (If retrieval is guided by a similarity measure regarding problem structures rather than attributes, mapping is performed simultaneously with retrieval).
- *Adaptation*: Adapt the example solution to the goal problem. (We will show, that this process can be performed purely syntactical for isomorphic problem structures. For non isomorphic problems additionally semantic information of the problem domain has to be regarded.)

[5]Tail-recursion has the structure $G(x) = g(boolop(x), op(x), G(minop(x)))$ If recursion terminates the result can directly be given. In more complex cases, as for example the clearblock problem, recursion depends on a stack. When recursion terminates, the result has to be used to calculate the expression obtained by each recursive call

- *Learning*: Construct a scheme which generalizes over the structure of example and goal problem.

In our approach we use analogical problem solving for the construction of RPSs from initial programs. We regard a current initial program as goal problem and search for a recursive generalization of this initial program. Already inferred RPSs are stored in a (partial) hierachical memory structere. How this structure is constructed is described below. To compare the goal problem with the RPSs stored in memory these RPSs (example solutions) have to be expanded to initial programs again (example problems). This "unfolding" (Burstall & Darlington, 1977) is preformed until the depth corresponds to that of the current initial tree. Similarity between goal problem and example problem is calculated by a measure for tree distance proposed by Lu (1979). This measure is a purely syntactical approach to compare structures and is extremely suitable for our purpose: initial programs are terms which can be represented by trees. Similarity between trees is calculated as sum of all operations which have to be performed to transform one tree into another tree. Three operations are defined: Substituting one node label by another (s), deletion of a node (d) and insertion of a node (i). The three types of operations can be associated with different weights. Similarity between two trees x and y is:

$$ds(x, y) = s * n_1 + d * n_2 + i * n_3$$

where n_i represents the frequency of performing one operation.[6]

If a tree can be transformed into another by unique substitutions only, the problems are isomorph. This is for example the case for the problems $factorial$ and sum given in figure 5a,b below. The substitutions are $\{1/0, mult/plus\}$.

Calculation of structural similarity between two initial programs provides simultaneously the mapping between both problems. Adaptation is easily achieved for isomorphic problems: The substitutions gained by the tree transformation can be applied to the RPS corresponding to the selected example problem. If the RPS for calculating $factorial$ is already known

$$fac(x) = g(equal0(x), 1, mult(x, fac(pred(x))))$$

the RPS for calculating the sum of x can be generated by substituting 1 by 0 and $mult$ by $plus$.

The new RPS is stored in memory. Example and goal RPS are generalized by variabilization of all symbols which occur at identical positions in both structures but are differently named. The generalized RPS is stored as parent node to example and goal RPS. This strategy provides that the memory gets more and more hierarchically organized with experience. The hierarchical organization additionally prevents retrieval to become inefficient when the number of RPSs in memory increases: Search for a suitable example problem starts with the roots of the RPSs hierarchies. Only the RPS structure where the root RPS has the highest similarity to the goal problem is further explored. In a next step, all children of the selected root RPS are exlored. Afterwards, only the branch with the highest similarity to the example RPS is regarded and so on. Note, that similarity can get lower when proceeding to the more special RPSs further beneath in the hierarchical

[6]Currently we are exploring an extension of this similarity measure where information about types and arity of function symbols (nodes) are taken into acount.

Table 2: Mapping of the clearblock and the factorial problem (mapping $topof$ to the identity function $id(x) = x$ corresponds to deletion of a node)

Clearblock	Factorial
x	x
s	1
cleartop	equal0
puttable	mult
topof	pred
topof	id

$\rightarrow$ non unique mapping
$\leftarrow$ surjective mapping

structure. In this case a generalized RPS is used for analogical problem solving. That is, the new problem is not solved by analogy but by application of an abstract problem scheme.

Finally we will give an example of analogical problem solving with non isomorphic structures. We will demonstrate, how a RPS for calculating the factorial function (see figure 5a) can be constructed from the RPS of the clearblock problem (see figure 5c) by analogical transfer. That is, we show analogical transfer from a visually concrete to an mathematical abstract problem domain. Figure 5 gives the initial programs represented as trees. Note that the tree in figure 5c corresponds to the tree of the clearblock problem given in figure 3. We use the more complex term representation here because this kind of representation is input to structure mapping with the tree distance. Table 2 gives the transformation which have to be performed. Additionally to substitution of node labels, two deletions of $topof$ nodes have to be performed. The two trees are not only not isomorphical, but there is even no unique mapping from clearblock to factorial. For analogical transfer it has to be determinded, at which position in the clearblock-RPS $topof$ has to be replaced by $pred$ and at which position $topof$ has to be deleted (i.e. replaced by the identity function). This can be done by regarding the context of the trees when calculating the transformation: $topof$ is deleted in the first argument of *puttable* only, therefore it will deleted in the first argument of *mult*. The both non isomorphic RPSs are:

G(x,s)	= g(cleartop(x),	s,	puttable(topof(x),	G(topof(x),s)))
F(x)	= g(equal0(x),	1,	mult(x,	F(pred(x)))).

An alternative strategy to "direct" adaptation is to try to generate an isomorphic relation between problems by transforming the example problem with the help of background knowledge (O'Hara, 1992). We represent term rewrite rules to represent background knowledge. If we use

$$cleartop(x) \rightarrow notexists(topof(x))$$

that is, we look at the problem in a new way, we can gain at least a partial isomorphism to the factorial problem (see figure 5a and 5d).

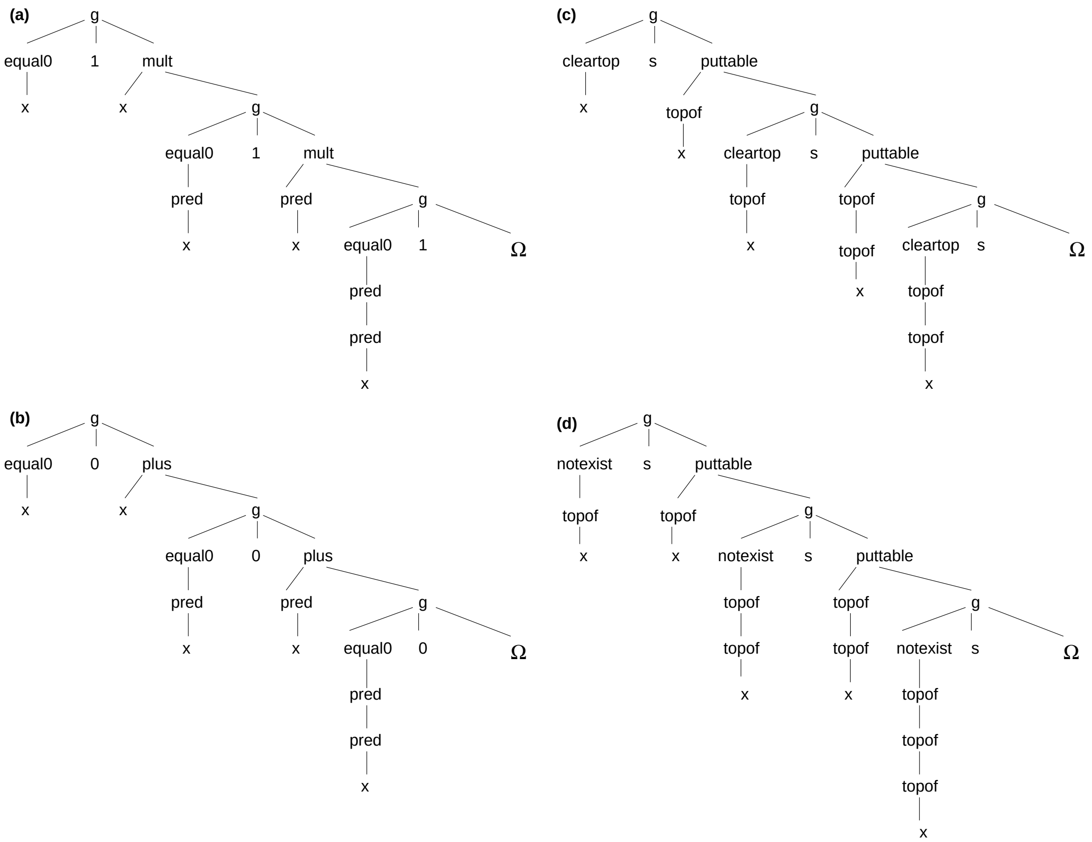

Figure 5: Initial programs represented as term trees for factorial (a), sum (b), clearblock (c) and transformed clearblock (d)

If we omit the special case of $x = 0$ (calculating only the factorial of $x - 1$) and allow for a constant as parameter, the problems are isomorphic:

$$\begin{array}{llll} \text{G(x,s)} & = \text{g(notexists(topof(x)),} & \text{s,} & \text{puttable(topof(x),} \quad \text{G(topof(x),s)))} \\ \text{F(x,1)} & = \text{g(equal0(pred(x)),} & \text{1,} & \text{mult(pred(x),} \quad \text{F(pred(x),1))).} \end{array}$$

This solution can be repaired furtheron by rewriting the RPS for the factorial problem:

$$\begin{array}{llll} \text{F(x)} & = \text{F'(x,x)} & & \\ \text{F'(x,y)} & = \text{g(equal0(pred(x)),} & \text{y,} & \text{mult(pred(x),} \quad \text{F'(pred(x),y))).} \end{array}$$

The RPS $F'(x, y)$ calculates the factorial by multiplying y (instantiated with x) with the recursively calculated factorial of $x-1$. Thereby a isomorphical relation between the clearblock problem and the factorial problem has been constructed.

While our first strategy of "direct" adaptation remains purely syntactical, the second strategy of problem transformation relies heavily on knowledge about the problem domain. While there is evidence that people can use structural similar but not isomorph structures for analogical transfer (see Pirolli and Anderson (1985), Schmid and Kaup (1995)) there is little knowledge about the strategies, people employ to adapt solutions in such cases. We believe, that both strategies are possible. But this is clearly a point for further investigation.

6 Acquisition of motor and process control skills

In this section we describe how the acquisition of motor and process control skills can be integrated in our framework. In Artificial Intelligence as well as in Cognitive Psychology, the acquisition of problem solving skills and the acquisition of motor or process control behavior is usually explored within different approaches. Problem solving skills ("cognitive skills") are typically described by production rules or similar approaches as described in the sections above. Control behavior on the other hand is typically described by neural or cybernetic circuits ("control programs"). Acquistion of control skills is often modelled by reinforcement learning (Dean, Basye, & Shewchuk, 1993), sometimes by artificial neural nets, for example with backpropagation as learning procedure (Nguyen & Widrow, 1989). Well known approaches to this domain in Artificial Intelligence are BOXES (Michie & Chambers, 1968), CART (Connell & Utgoff, 1987) and ASE/ACE (Barto, Sutton, & Anderson, 1983), mainly concerned with maintaining a stable system state. We do not argue that this kind of "low level" skill learning (in most cases leading to the "credit assignment" problem) is not important for modelling human skill acquisition. But we are interested in this paper in control behavior as analogous to problem solving, employing problem solving together with machine learning techniques.

If we look at motor/control behavior skills from the perspective of problem solving, we identify the following similarities and differences: In problem solving, an intelligent system manipulates problem states to reach a predefined goal in an otherwise static environment. In motor or process control behavior a system interacts with another system to reach or maintain a goal state. States and operations might describe aspects of the controlling system itself and of its environment (in behavior control tasks as for example riding a bicycle) or of an other system (in process control as for example performing a chemical synthesis task). Problem states are in general real valued parameters describing the current characteristics of the system which is to be controlled (as values for heat and pressure) or the controlling system itself (as position and speed). The controlling system is provided with a set of operators which can be continuous (as move for n meters, raise temperature to n degrees). But the conditions for operator applications may be not known, that is, there may be no or only an incomplete domain model. Additionally, operator applications are not the only way to change system states, the controlling system might have to deal with additional influcence factors from the environment (as wind), i.e. so called open systems. Problem solving is done by operator applications whereby the system is moving along trajectories in some (in most cases continuous) state space in or into a goal area. Whereas in classical problem solving, the complete problem space could be explored, this is in general not possible in the continous probem spaces we deal with in motor/process control.

Acquisition of motor/control behavior skill can be regarded as similar to the construction of initial programs from problem solving experience (see section 3). In a first step, a representative set of input states ("training data") has to be generated. Then, a problem solving algorithm determines the optimal control action for each state by means of an empirical evaluation function. Other than in discrete problem spaces, here the goal state usually is not a point, but an area in the parameter space (i. e. continuous problem space). Müller and Wysotzki (1995) employ a decision tree technique to split the state space automatically into subareas with unique control actions (CAL 5, see Appendix). The resulting decision tree represents a control program: It can be read as a nested conditional expression, where a state is tested in which area of the parameter space it falls and the appropriate control action is executed, giving the next state and so on, i.e., this program can be applied recursively, where the i-th input is the result of the application of an control operation to the input $i-1$.

In the following we will describe this kind of generalization (over problem states) in more detail. As an real world example we use the roll axis stabilization (as a reaction to disturbances) of a communication satellite in orbit with respect to its position to the earth axis (Müller & Wysotzki, 1995)[7]. The position manoeuvers of the satellite are carried out by thruster torques delivering a set of discrete control actions the application of which consumes fuel. The control task is to bring back the satellite axis into a certain small acceptable target interval of its attitude angle consuming as little fuel as possible (i.e. performing a small number of control actions) after distortions.

The state space of the process is defined by the attitude angle φ of the satellite and its rate $\dot{\varphi}$. The target area of control is some region around $\varphi = 0$ and $\dot{\varphi} = 0$ (some maximal value of $|\varphi|$ is given which must not be exceeded). There are additional variables describing oscillations of the two solar generators of the satellite which influence the process states, considered as noise.

In a first step a training set of n randomly selected points in the two dimensional state space X is constructed, which are interpreted as distorted states as compared with the goal state. Each input is a tuple $m_j = (\varphi, \dot{\varphi}) \in X$. Additionally there is a set of control actions $P_f = \{f_1, ... f_q\}$ and an evaluation function $F : X \rightarrow \mathcal{R}$, which maps the state space into the set of real numbers. Part of F is the minimal distance to the target area. For each input (i.e. distorted state) the local optimal control action is calculated, using gradient descent (hill climbing) on F: All available control actions are applied to a given input state and thereby a set of following states is constructed (the next state is computed using a simulation program since of course it is not possible to experiment with the satellite, for detaills see Müller and Wysotzki (1995)). The (unique) action f_{opt} which leads to a following state with minimal value of F is associated with the input state. For the satellite problem local optimal actions result in global optimal trajectories (see below).

The pairs (m_j, f_{opt}) are now used to construct a control program. Here the classification algorithm CAL5 (Unger & Wysotzki, 1981; Müller & Wysotzki, 1994) is used, which constructs decision trees by automatic calculation of optimal discretizations of continuous parameter values (see figure 6).

The decision tree classifies system states m_j with respect to the control action f which should be applied to each state. It can be read as a conditional program ("IF φ is less than -1.7 THEN use thruster torque T_c ..."). This program generalizes over the presented input states from which it was constructed, segmenting the whole state space in regions with associated optimal control actions. Since we assume that there are no "obstacles" (e.g. instability regions) in the state space to be avoided by the control mechanism, the tree contains implicitly almost optimal control trajectories from a given input state to the goal area. This is due to the fact that our problem solving algorithm is greedy, that is we have a local criterion which guarantees global optimality (there are no "obstacles" causing backtracking in the problem space). It has been shown that the generalization

[7]This is an application conducted by Wolfgang Müller and Fritz Wysotzki of the Fraunhofer Institute for Information and Data Processing, Branch Lab for Process Optimization (EPO), Berlin, as part of the project WISCON, supported by the German Ministry of Science and Technology.

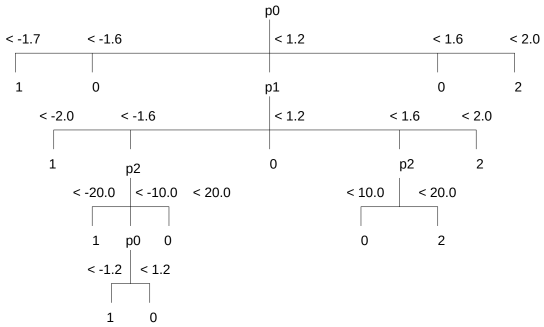

Figure 6: Program for controlling the behavior of a satellite represented as decision tree ($p_0 = \varphi$, $p_1 = \dot{\varphi}$, p_2 = oscillation of one solar generator; 0 = no control action, $1 = +T_c$, $2 = -T_c$)

error during tree learning does not much influence the optimality of the created control trajectories. While the initial steps - problem solving and construction of the control program - are to be time consuming processes, the resulting program can be used for efficient realtime process control (see figure 7).

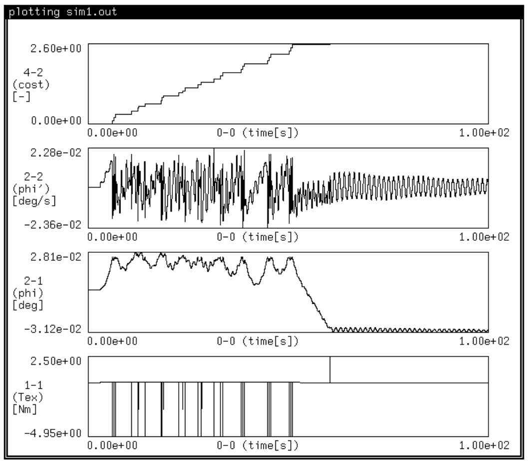

Figure 7: Plot of real time control for the satellite

7 Discussion

We have proposed to look at skill acquisition from the viewpoint of program synthesis. Thereby we gain *(1)* a unifying view for acquisition of cognitive as well as motor/ process behavior and *(2)* have a sound theoretical background in theoretical computer science.

Induction and analogical inference of recursive program schemes can be viewed as a general approach to the acquisition of problem solving skills as proposed in section 1. If we take a more restricted view on recursive program schemes as functional programs, our approach can be seen as a contribution to research on the acquisition of programming skills: In a first step it describes, how straight forward programs are constructed by problem solving on desired inputs to a functional program. Problem solving operators can be restricted to represent builtin functions of a given programming language. The initial program for factorial (see figure 5a) can be constructed by exploring what to do for calculating the factorial of $n = 3$ in a straight forward way using the basic definitions for the builtin function *mult* as operator and the boolean function *equal*0 as predicate to describe inputs (i.e. problem states). Inductive program synthesis can be viewed as an "all purpose inference strategy" to construct recursive programs. Our approach to analogical problem solving and learning describes, how recursive programs can be constructed by analogical transfer and how a programming novice gains experience of problem structures in the domain of recursive programming by generalizing over example problems.

Taking this more restricted perspective, our approach is not only a contribution to cognitive modeling but also to automatic programming and knowledge based software engineering (Lowry & McCarthy, 1991). Enriching classical approaches to inductive program synthesis by problem solving and analogical reasoning and learning can result in more powerful systems. Usually, input in inductive synthesis systems is a set of input/output examples which have to be reformulated by background knowledge about datastructures and builtin operations to construct initial programs (Summers, 1977). We propose an alternative strategy where initial programs can be constructed from example inputs alone, constructing corresponding outputs by problem solving. Introducing a memory for RPSs for programming by analogy is considered a long time as means for increasing effectivity and power of inductive program synthesis systems (Manna & Waldinger, 1975), but rarely realized (Sadohara & Haraguchi, 1995). In the domain of automatic theorem proving there is already some evidence that analogical reasoning can increase system performance (Melis, 1995). Furthermore, enriching builtin functions by the set of already synthesized RPSs opens the way to construct more complex programs.

Acknowledgements
Implementation of the system is done by Mark Müller, incorporating work which was done in two studentical projects on the Berlin University of Technology.

References

Anderson, J. (1986). Knowledge compilation: A general learning mechanism. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine learning - an artificial intelligence approach* (Vol. 2, p. 289-310). Tioga

Anderson, J. (1993). *Rules of the mind.* Hillsdale, NJ: Lawrence Erlbaum

Anderson, J., & Thompson, R. (1989). Use of analogy in a production system architecture. In S. Vosniadou & A. Ortony (Eds.), *Similarity and analogical reasoning* (p. 267-297). Cambridge University Press

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

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science*, *13*, 467-505

Anzai, Y., & Simon, H. (1979). The theory of learning by doing. *Psychological Review*, *86*, 124-140

Barto, A. B., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, *13*(5)

Burstall, R., & Darlington, J. (1977). A transformation system for developing recursive programs. *JACM*, *24*(1), 44-67

Connell, M. E., & Utgoff, P. E. (1987). Learning to control a dynamic physical system. In *Proceedings AAAI-87* (pp. 456–460). Seattle

Cornuejlos, A. (1997). Analogy as minimization of description length. In N. Nakhaeizadeh & C. Taylor (Eds.), *Machine learning and statistics. The interface* (p. 321-335). New York: Wiley

Dean, T., Basye, K., & Shewchuk, J. (1993). Reinforcement learning for planning and control. In S. Minton (Ed.), *Machine learning methods for planning* (p. 67-92). Morgan Kaufmann

Falkenhainer, B., Forbus, K., & Gentner, D. (1989). The structure mapping engine: Algorithm and example. *Artificial Intelligence*, *41*, 1-63

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

Gold, E. (1967). Language identification in the limit. *Information and Control*, *10*, 447-474

Green, C. (1969). *Application of theorem proving to problem solving* (Tech. Rep.). IJCAI 1.

Holland, J., Holyoak, K., Nisbett, R., & Thagard, P. (1986). *Induction - processes of inference, learning, and discovery.* Cambridge, MA: MIT Press

Holyoak, K. J., & Thagard, P. (1989). Analogical mapping by constraint satisfaction. *Cognitive Science*, *13*, 295-355

Klahr, D., Langley, P., & Neches, R. (1987). *Production system models of learning and development.* Cambridge, MA: MIT Press

Klix, F., & Rautenstrauch-Goede, K. (1967). Struktur- und Komponentenanalyse von Problemlöseprozessen. *Zeitschrift für Psychologie*, *174*, 167-193

Lowry, M. L., & McCarthy, R. D. (1991). *Autmatic software design.* Cambridge, Mass.: MIT Press

Lu, S. (1979). A tree-to-tree distance and its application to cluster analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *PAMI-1*(2), 219-224

Manna, Z., & Waldinger, R. (1975). Knowledge and reasoning in program synthesis. *Artificial Intelligence*, *6*, 175-208

Melis, E. (1995). A model of analogy-driven proof-plan construction. In C. S. Mellish (Ed.), *Proceedings of the fourteenth international joint conference on artificial intelligence* (pp. 182–189). San Mateo: Morgan Kaufmann

Michie, D., & Chambers, R. A. (1968). Boxes: An experiment in adaptive control. In E. Dale & D. Michie (Eds.), *Machine intelligence* (Vol. 2, p. 137-152). Edinburgh: Oliver and Boyd

Müller, W., & Wysotzki, F. (1994). Autmatic construction of decision trees for classification. In K. Moser & K. M. Schader (Eds.), *Annals of operation research* (Vol. 92, p. 231-247). Wijdenes, The Netherlands: J. C. Baltzer AG Science Pub.

Müller, W., & Wysotzki, F. (1995). Automatic synthesis of control programs by combination of learning and problem solving methods. In N. Lavrač & S. Wrobel (Eds.), *Machine learning: ECML-95 (Proc. European Conf. on Machine Learning, 1995)* (pp. 323 – 326). Berlin,: Springer

Newell, A. (1991). *Unifyied theories of cognition.* Cambridge, MA: Cambridge University Press

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, NJ: Prentice Hall

Nguyen, D., & Widrow, B. (1989). The truck backer upper: an example of self-learning in neural networks. In *Proc. IJCNN* (Vol. 2, p. 357-363).

Nilsson, N. J. (1980). *Principles of artificial intelligence.* New York: Springer

Novick, L. R. (1988). Analogical transfer, problem similarity, and expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *14*, 510-520

Novick, L. R., & Holyoak, K. J. (1991). Mathematical problem solving by analogy. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *17*(3), 398-415

O'Hara, S. (1992). A model of the redescription process in the context of geometric proportional analogy problems. In *Analogical and inductive inference, int. workshop aii '92, dagstuhl castle, germany* (Vol. LNAI 642, p. 268-293). Heidelberg: Springer

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

Reed, S. K., & Bolstad, C. (1991). Use of examples and procedures in problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *17*(4), 753-766

Rosenbloom, P. S., & Newell, A. (1986). The chunking of goal hierarchies: A generalized model of practice. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning - an artificial intelligence approach* (Vol. 2, p. 247-288). Morgan Kaufmann

Rumelhart, D., & Norman, D. (1978). Accretion, tuning and restructuring: Three modes of learning. In J. Cotton & R. Klatzky (Eds.), *Semantic factors in cognition* (p. pp. 37-53). . Hillsdale, NJ: Erlbaum

Rumelhart, D. E., & Norman, D. A. (1981). Analogical processes in learning. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (p. 335-360). Hillsdale, NJ: Lawrence Erlbaum

Sadohara, K., & Haraguchi, M. (1995). Analogical logic program synthesis from examples. In N. Lavrac & S. Wrobel (Eds.), *Proceedings of the 8th European Conference on Machine Learning (ECML-95)* (Vol. 912, p. 232-244). Springer

Schmid, U., & Kaup, B. (1995). Analoges Lernen beim rekursiven Programmieren (Analogical learning in recursive programming). *Kognitionswissenschaft*, *5*, 31-41

Schmid, U., & Wysotzki, F. (1996). Fertigkeitserwerb durch induktive Programmsynthese und generalisiertes Planen. In W. Dilger, M. Schlosser, J. Zeidel, & A. Ittner (Eds.), *Proceedings of FGML-96 (19.-21.8.96, TU Chemnitz)* (p. 106-111). Chemnitz

Schmid, U., & Wysotzki, F. (1996). Induction of recursive program schemes. In C. Nedellec & C. Rouveirol (Eds.), *Proceedings of ECML-98* (p. 228-240). Springer

Shavlik, J. W. (1990). Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, *5*, 39-70

Shell, P., & Carbonell, J. (1989). Towards a general framework for composing disjunctive and iterative macro-operators. In *11th IJCAI-89.* Detroit, MI

Simon, R. J. (1970). Encoding effects on complex problem solving. *Journal of Experimental Psychology*, *83*, 227-231

Summers, P. D. (1977). A methodology for LISP program construction from examples. *Journal ACM*, *24*(1), 162-175

Unger, S., & Wysotzki, F. (1981). *Lernfähige Klassifizierungssysteme.* Berlin: Akademie-Verlag

Vosniadou, S., & Ortony, A. (1989). Similarity and analogical reasoning: A synthesis. In S. Vosniadou & A. Ortony (Eds.), *Similarity and analogical reasoning* (p. 1-17). Cambridge: Cambridge University Press

Wysotzki, F. (1983). Representation and induction of infinite concepts and recursive action sequences. In *Proceedings of the 8th IJCAI.* Karlsruhe

Wysotzki, F. (1987). Program synthesis by hierarchical planning. In P. Jorrand & V. Sgurev (Eds.), *Artificial intelligence: Methodology, systems, applications* (p. 3-11). Amsterdam: Elsevier Science

Appendix

The decision tree algorithm CAL 5

see Unger and Wysotzki (1981), pp. 42 and Müller and Wysotzki (1994) for more details.

Given:

- A list of training examples M. Each example is represented by an attribute (feature) vector $(x_1, ... x_n)$ associated with a class (optimal operator) k.
  A new attribute x to be introduced into the decision tree can be selected either by an predefined order (from x_1 to x_n) or by using an entropy measure method (selecting that attribute which discriminates best between classes).
- A decision threshold $S \leq 1$.

1. Introduction of the next attribute x_i.
2. Automatic construction of optimal intervals (discretization).
   (a) Take all values a_j of the attribute x_i which occur in the training examples. Order the values by the $\leq$ relation with minimal and maximal values of the range of the parameter x_i as lower and upper bound ($-\infty$ and $+\infty$ if no special restriction is known): $min < a_1 < a_2 ... < a_m < max$.
   (b) Construct intervals for each attribute value: $I = (a_j, a_{j+1}]$.
   Each interval is associated with the classes k_l which are associated with the examples where $x_i = a_j$. Each class is annotated by the frequency n_{k_l} with which it occurs (see figure below).
   (c) Recursion: To form intervalls on the x_i axis, collect samples from left to right, until a class decision can be made on a given level of confidence $1 - \alpha$.
   Let I be a current interval containing n samples of different classes k and let n_{k_l} be the number of samples belonging to class k_l. Then $q_{k_l} = n_{k_l}/n$ is an estimate of the conditional probability $p(k_l|x)$ in I, where x is the current node (sample at interval I).
      - The hypothesis:
        H1: There exists a class k_l in I with $p(k_l|x) \geq S$
      - is tested against the hypothesis:
        H2: For all classes k_l in i the inequation $p(k_l|x) < S$ holds on the given level of confidence $1 - \alpha$.

   An estimation at level $1 - \alpha$ yields a confidence interval $[d_1, d_2]$ for $p(k_l|x)$. That means, that in a sequence of samples the true value of the probability lies in $[d_1, d_2]$ with probability $1 - \alpha$.
   The hypotheses are tested in the following way:
      - **H1:** $d_1(k_l) > S$ (if the complete confidence interval lies above the predefined threshold S, then H1 is true)

- **H2:** $d_2(k_l) < S$ for all k_l occuring in intervall I (if for each class k_l in I the confidence interval lies below S, then H2 is true).

Decision on the dominance of a class in I:

i. If a class k_l exists, where H1 is true, then k_l dominates in I. The interval is closed. The corresponding path in the decision tree is terminated.

ii. If for all classes in I H2 is true, then no class dominates in I. The interval is closed. A new attribute has to be introduced (Step 1).

iii. Otherwise the interval I has to be extended bu collecting the next sample (merging the current interval $I = (a_j, a_{j+1}]$ with the succeeding interval $I' = (a_{j+1}, a_{j+2}]$. This corresponds to an increase of the statistics to obtain a decision for H1 or H2 at confidence level $1 - \alpha$.
If there are no more samples for extending I, a majority decision is performed.

3. Merging of intervals.
Adjacent intervals with the same class label are merged.

4. The algorithm terminates when all intermediate nodes have disappeared, that is, each leaf of the current decision tree represents a dominant class or a majority decision for a class.