

# A Cognitive Model of Learning by Doing

**Ute Schmid**

Dept. of Information Systems and Applied Computer Science  
Bamberg University, D-96045 Bamberg  
email: ute.schmid@wiai.uni-bamberg.de

February 21, 2005

## **Abstract**

In this paper an approach to learning cognitive skills from problem solving experience is presented – addressing some phenomena well known from human learning but seldom covered together in machine learning. The core of our approach is the acquisition of recursive program schemes (RPSs) by generalization-to-n over plans, using an inductive program synthesis technique. RPSs represent domain specific control knowledge, that is problem solving strategies for classes of problems. Because RPSs are abstract schemes, representing the sub-goal structure of a domain, they are suitable for analogical problem solving and learning.

## **1 Introduction**

If an expert system – brilliantly designed, engineered and implemented – cannot learn not to repeat its mistakes, it is not as intelligent as a worm or a sea anemone or a kitten.

- Oliver G. Selfridge, from *The Gardens of Learning*.

Find a bug in a program, and fix it, and the program will work today. Show the program how to find and fix a bug, and the program will work forever.

- Oliver G. Selfridge, in *AI's Greatest Trends and Controversies*

The most important building-stones for the flexible and adaptive nature of human cognition are powerful mechanisms of learning. On the low-level

end of learning mechanisms is stimulus-response learning, mostly modelled with artificial neural nets or reinforcement learning. On the high-level end are different principles of induction, i. e., generalizing rules from examples. While the majority of work in machine learning focusses on induction of concepts (classification learning, see Mitchell, 1997 for an introductory overview), our work focusses on inductive learning of cognitive skills from problem solving. While concepts are mostly characterized as *declarative* knowledge (know what) which can be verbalized and is accessible for reasoning processes, skills are described as highly automated *procedural* knowledge (know how) (Anderson, 1983).

Problem solving is generally realized as heuristic search in a problem space (Newell & Simon, 1972). In cognitive science most work is in the area of goal driven production systems (ACT, Anderson, 1993; SOAR, Newell, 1990), in AI different planning techniques are investigated (Graphplan, Blum & Furst, 1997, HSP Bonet & Geffner, 2001). In both frameworks, the definition of problem operators together with conditions for their application – i. e., production rules – is central. Matching, selection, and application of operators is performed by an interpreter (control strategy): the preconditions of all operators defined for a given domain are matched against the current data (problem state), one of the matching operators is selected and applied on the current state. The process terminates if the goal is fulfilled or if no operator is applicable.

Skill acquisition is usually modelled as composition of predefined primitive operators as result of their co-occurrence during problem solving – i. e., learning by doing. This is true in cognitive modelling (knowledge compilation, Anderson & Lebière, 1998; operator chunking, Rosenbloom & Newell, 1986) as well as in AI planning (macro learning, Minton, 1985; Veloso et al., 1995). Acquisition of such “linear” macros results in a reduction of search, because now composite operators can be applied instead of primitive ones. In cognitive science, operator-composition is viewed as responsible mechanism for acquisition of automatisms and the main explanation for speed-up effects of learning (Anderson, Conrad, & Corbett, 1989).

In contrast, in AI planning, learning of domain specific control knowledge, that is, learning of problem solving *strategies*, are investigated as an additional mechanism (Martín & Geffner, 2000). One possibility to model acquisition of control knowledge is learning of “loop” macros (Shell & Carbonell, 1989; Shavlik, 1990). Learning a problem solving strategy ideally eliminates search completely because the complete sub-goal structure of a

problem domain is known. For example, a macro for a one-way transportation problem as *rocket* (Veloso & Carbonell, 1993) represents the strategy that *all* objects has to be loaded before the rocket moves to its destination. There is empirical evidence, for example in the Tower of Hanoi domain, that people can acquire such kind of knowledge (c. f., Anzai & Simon, 1979).

An alternative approach to problem solving as search is problem solving by analogy: For a given problem (target), a similar problem with known solution (source) is retrieved from memory and this solution is adapted to the new problem. Analogical problem solving is either modelled as transfer of a solution based on mapping the structures of source and target (Falkenhainer, Forbus, & Gentner, 1989) or as replaying the source solution in the new context (Carbonell, 1986; Veloso, 1994). Analogical learning is described as generalization over the common structure of example and goal problem. While there is empirical evidence that humans acquire generalized knowledge from analogical problem solving (Novick & Holyoak, 1991; Schmid & Kaup, 1995), only few models of analogical reasoning address learning (Anderson & Thompson, 1989).

A draw-back of psychological models of and AI approaches to learning by analogy is, that the existence of scheme-like representation of problems is mostly presupposed. On the other hand, it is claimed that analogical problem solving is a strategy used, if the (cognitive) system has *no* general knowledge about a problem domain (Novick, 1988). There is next to no work on how initial, problem specific schemes are acquired (Rumelhart & Norman, 1981).

We are working on a technique for inductive synthesis of recursive program schemes which we propose as suitable to overcome some limitations of approaches to learning by doing and learning by analogy addressed above. In the following, we will first give an overview of our approach. Afterwards we will show: (1) how initial experience about a problem domain can be learned from planning, and (2) how this initial experience can be generalized to domain specific problem solving strategies. Because we represent such strategies as recursive program *schemes*, our method simultaneously can be used to model the acquisition of problem specific schemes which (3) can be used and further generalized in analogical problem solving.

Please note, that the formal background of our approach together with the algorithms and evaluations are presented in detail elsewhere (Schmid, 2003). In this paper we focus on how this approach can be used to model some aspects of skill acquisition.

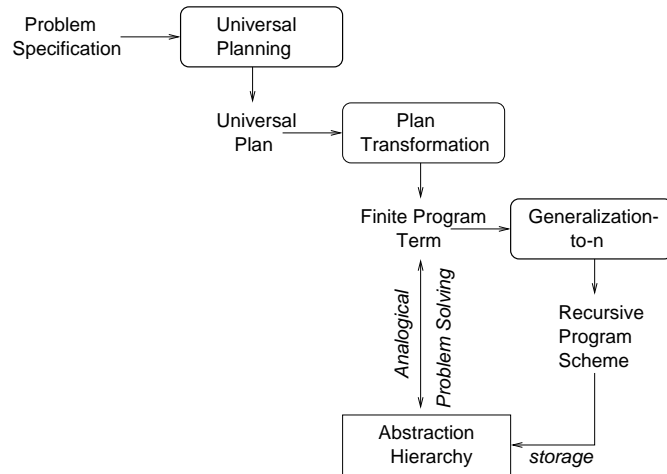


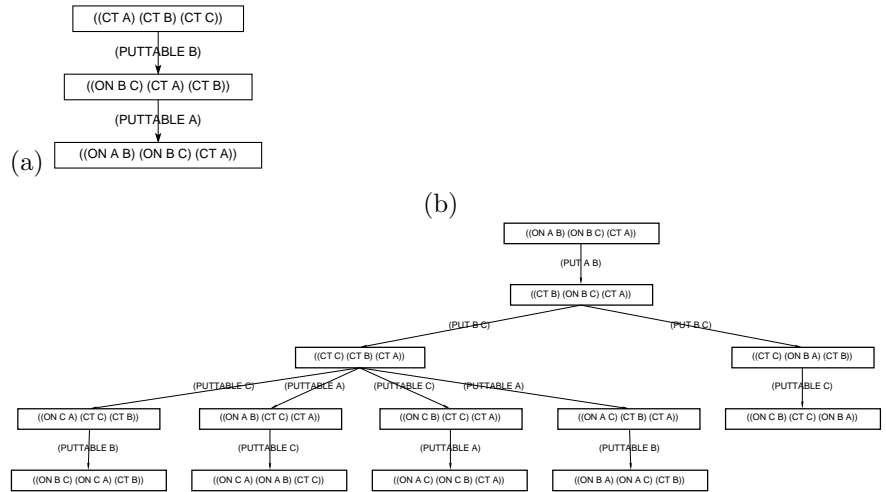
Figure 1: Components of IPAL

## 2 A Framework for Skill Acquisition

The general architecture of our system IPAL is given in figure 1: Input is some problem specification (pre-defined primitive operators, problem solving goal). The system explores the problem, that is, tries to find operator sequences for transforming some initial states into a state fulfilling the goals by means of search (planning). This experience is integrated in a finite program term (“initial program”), corresponding roughly to a set of linear macros as discussed above. The initial program is generalized to a recursive program scheme (RPS), i. e., the system infers a domain specific control strategy which simultaneously represents the (goal) structure of the current domain. This generalization-to-n step is realized by an inductive program synthesis technique. Alternatively, after initial exploration, a similar problem might be retrieved from memory. That is, the system recognizes that the new problem can be solved with the same strategy as an already known problem. In this case, a further generalized scheme, representing the abstract strategy for solving both problems is learned.

We will illustrate this three steps of learning with a simple blocks-world example used throughout the paper (see fig. 2): To reach the goal, that block  $C$  has a clear top, all blocks lying above  $C$  have to be put on the table. This can be done by applying the operator  $puttable(x)$  if block  $x$  has



Figure 3: Universal Plans for *clearblock* and *tower*

a clear top. The initial program represents the experience with three initial states as a nested conditional expression. This primitive behavioral program is generalized over recursive enumerable problem spaces: a strategy for clearing a block in  $n$ -block problems is extrapolated and interpreted as recursive program scheme. This scheme represents not only the solution strategy for unstacking towers of arbitrary height but for all structural identical problems. That is, experience with a blocks-world problem can for example be used to solve a numerical problem which has the same structure by re-interpreting the meaning of the symbols of an RPS.

### 3 Generalization over Problem States

For the first step of learning we use an approach of universal planning (Schoppers, 1987; Schmid & Wysotzki, 2000) to explore a problem domain with small complexity (e. g., a blocks-world with only 3 blocks). In contrast to classical planning, an universal planner “simultaneously” generates plans for *sets* of initial states.

Planning starts with a set of top-level goals. For the simple *clearblock* problem, the goal is *cleartop(C)*. For the problem of building a tower of

alphabetically sorted blocks, the goal could be  $\{on(A,B), on(B,C)\}$  – i. e., a conjunction of possibly interdependent goals. For a given set of operators, the planner constructs recursively all problem states which are legal predecessors of a current state. For the *clearblock* problem, we have only one operator

*puttable*( $x$ )

**PRE:** *cleartop*( $x$ ), *on*( $x,y$ )

**ADD:** *cleartop*( $y$ )

**DEL:** *on*( $x,y$ ).

In general, an operator is defined by preconditions (PRED) and effects (ADD, DEL). For *forward* operator application, the current state and the preconditions have to match. The operator is instantiated accordingly (and now called “action”) and applied to the current state. The state is transformed into a successor state by adding the literals specified in ADD and deleting the literals specified in DEL. For *backward* application of an instantiated operator  $o$ , the current state  $S$  and the ADD-List have to match and a predecessor state is constructed by subtracting ADD and adding PREC and DEL:  $Res^{-1}(S,o) = S \setminus ADD \cup (DEL \cup PRED)$ . For example, if we have the state  $\{on(B,C), cleartop(B), cleartop(A)\}$ , backward application of *puttable*( $A$ ) results in  $\{on(A,B), on(B,C), cleartop(A)\}$ .

A universal plan is generated by backward operator applications starting with a state fulfilling the top-level goals. For each current state (node in the planning tree), the plan is expanded by all legal predecessors which are not already contained in the plan. That is, the plan consists of all *optimal* plans. The resulting universal plans for *clearblock* and *tower* are given in figure 3. Modelling the exploration of a problem by universal planning is clearly not psychologically plausible. Alternatively, single plans could be generated sequentially and stepwise integrated into one structure.

For generalization over the structure of a plan, i. e. for inferring a problem scheme representing a domain specific solution strategy, the universal plan has to be transformed into an initial program (as given in fig. 2). The crucial aspect of transformation is to replace constants by “constructive” expression. In terms of theory of programming this corresponds to the definition of a constructive data type.

For example, for blocks-world problems, we can define  $on(x,y) \equiv topof(y) = x$ . Following the ordering of states given in the plan, we can replace

$B \rightarrow \text{topof}(C)$  and  $A \rightarrow \text{topof}(\text{topof}(C))$  and interpret the remaining constant  $C$  as variable. Using only relevant predicates (the top-level goals and all literals inserted by ADD), the *clearblock* program corresponds to the nested conditional expression given in figure 2. Each condition-expression pair of this initial program corresponds to a linear macro or generalized production rule:

$$\begin{aligned} & \text{IF } \text{cleartop}(x) \text{ THEN } s \\ & \text{IF } \text{cleartop}(\text{topof}(x)) \text{ THEN } \text{puttable}(\text{topof}(x),s) \\ & \text{IF } \text{cleartop}(\text{topof}(\text{topof}(x))) \text{ THEN } \text{puttable}(\text{topof}(x), \\ & \quad \text{puttable}(\text{topof}(\text{topof}(x)),s)). \end{aligned}$$

The symbol  $s$  represents the current state – it is a so called situation variable (Manna & Waldinger, 1987) (note, that operators get  $s$  as an additional argument).

## 4 Induction of Problem Schemes

For the second step of learning we use a generalization-to-n technique, as usual in inductive program synthesis (Shavlik, 1990; Schmid & Wysotzki, 1998; Kitzelmann, Schmid, Mühlfordt, & Wysotzki, 2002). Some initial experience with a problem domain, represented as initial program, can be generalized to a recursive program scheme, representing the sub-goal structure of a problem and thereby a strategy for operator-application.

Generalization-to-n can be realized by purely syntactical pattern-matching. We demonstrate the method again with the *clearblock* problem. The initial program given in figure 2 is our input to inductive program synthesis:

$$\begin{aligned} \mathcal{G} = & \text{if } \text{cleartop}(x) \text{ then } s \text{ else } \text{puttable}(\text{topof}(x), \\ & \text{if } \text{cleartop}(\text{topof}(x)) \text{ then } s \text{ else} \\ & \text{puttable}(\text{topof}(\text{topof}(x)), \\ & \text{if } \text{cleartop}(\text{topof}(\text{topof}(x))) \text{ then } s \text{ else } \Omega)). \end{aligned}$$

The symbol  $\Omega$  (denoting the “undefined”) indicates, that the system has gained no experience for the case that the second block on a block  $x$  is not clear. Generalization will now be performed 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 occurs

repeatedly with a substitution  $\Theta$  of variables  $v$  by terms  $t$ . To this aim we decompose  $\mathcal{G}$  in a sequence of terms  $G^{(i)}$  with  $G^{(i+1)} = tr(G_{\Theta}^{(i)})$ . In terms of theoretical computer science that is, we interpret  $\mathcal{G}$  as an element of a Kleene sequence (linear expansion) belonging to some unknown recursive program. By definition  $G^{(0)} = \Omega$  holds. Now we can rewrite  $\mathcal{G}$  as sequence

$$\begin{aligned} G^{(0)} &= \Omega \\ G^{(1)} &= \text{if clear}top(x) \text{ then } s \text{ else } \text{puttable}(topof(x), \Omega) \\ G^{(2)} &= \text{if clear}top(x) \text{ then } s \text{ else } \text{puttable}(topof(x), \\ &\quad \text{if clear}top(topof(x)) \text{ then } s \text{ else} \\ &\quad \text{puttable}(topof(topof(x)), \\ &\quad \quad \Omega)) \\ G^{(3)} &= \mathcal{G} \end{aligned}$$

with  $tr = \text{if clear}top(x) \text{ then } s \text{ else } \text{puttable}(topof(x), m)$  and  $\Theta = \{x/topof(x)\}$ . Symbol  $m$  marks the place where a sub-term can be inserted into the structure. We can rewrite the sequence by

$$\begin{aligned} G^{(0)} &= \Omega \\ G^{(1)} &= \text{if clear}top(x) \text{ then } s \text{ else } \text{puttable}(topof(x), \\ &\quad G_{\{x/topof(x)\}}^{(0)}) \\ G^{(2)} &= \text{if clear}top(x) \text{ then } s \text{ else } \text{puttable}(topof(x), \\ &\quad G_{\{x/topof(x)\}}^{(1)}) \\ G^{(3)} &= \text{if clear}top(x) \text{ then } s \text{ else } \text{puttable}(topof(x), \\ &\quad G_{\{x/topof(x)\}}^{(2)}) \end{aligned}$$

and extrapolate

$$G^{(i+1)} = \text{if clear}top(x) \text{ then } s \text{ else } \text{puttable}(topof(x), G_{\Theta}^{(i)})$$

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

$$\begin{aligned} G(x, s) &= \text{if clear}top(x) \text{ then } s \\ &\quad \text{else } \text{puttable}(topof(x), G(topof(x), s)). \end{aligned}$$

Induction of generalized structures from examples is a fundamental characteristic of human intelligence as for example proposed by Chomsky as “language acquisition device” (Chomsky, 1959). This ability to extract general rules from some initial experience is captured in the presented technique of inductive program synthesis.

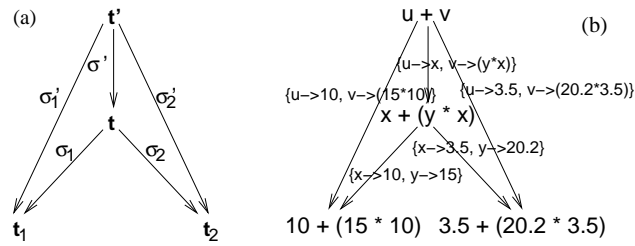


Figure 4: Mapping via abstraction and instantiation

## 5 Learning by Analogy

For the third step of learning we apply a method of generalizing over program schemes which is based on analogical transfer (Hasker, 1995; Schmid, Burghardt, & Wagner, 2003). 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 an RPS with a certain recursive structure exists, a structural isomorphic (or structural similar) problem can be solved by analogical transfer omitting the effort of inducing an RPS from scratch.

A current initial program (target problem) might be solvable by means of a strategy already learned, i. e., an RPS represented in memory. How a suitable RPSs can be found in memory is discussed at the end of this section.

The crucial step in analogy is to find a mapping from the base to the target problem. Typically in cognitive models, mapping is performed *directly* from objects of the base domain to objects of the target domain (Gentner, 1983; Falkenhainer et al., 1989). An alternative view to mapping is this (see Fig. 4.a): first, identify the common structure ( $t$ ) of base ( $t_1$ ) and target ( $t_2$ ), second, calculate the mapping of objects of the base to objects of the target *via* the common structure.

In structure mapping as well as in mapping via abstraction, the greatest common sub-structure of base and target must be identified. But in

mapping via abstraction, the common structure is not forgotten but represented explicitly. Instead of mapping one object directly onto another, it is mapped via the *role* it is playing in the common structure. For example, the number “0” can play the role of a neutral element in addition. Mapping via abstraction in general is more efficient than direct mapping because there is no need to try out different possible mappings. In the LISA model (Hummel & Holyoak, 1997) mapping is constrained by semantic features associated with the objects. In anti-unification, the constraints are given by the problem structure, or, in other words, by the *functional role* the objects play in a given structure.

The mapping procedure just described is known as calculating the most special generalization (Plotkin, 1969) or as anti-unification (Reynolds, 1970). It is important, that the *most special* generalization is calculated. Only then all structural commonalities are captured in the abstract representation. Formally, a generalization of  $t_1$  and  $t_2$  can be characterized as most specific, if for each other generalization  $t'$  holds, that there exists a substitution  $\sigma'$  such that  $t'\sigma' = t$  (see Fig. 4.a).

Anti-unification (AU) is defined for clauses (that is logical formulae) as well as for terms. In the following, we only deal with terms. For further illustration of AU, look at the following example (see Fig. 4.b):

$$t_1 = 10 + (15 \cdot 10) \text{ and } t_2 = 3.5 + (20.2 \cdot 3.5).$$

The most special generalization is  $t = x + (y \cdot x)$  with  $\varphi = \{(10, 3.5) \mapsto x, (15, 20.2) \mapsto y\}$  and it holds that  $t_1 = t\sigma_1$  for  $\sigma_1 = \{x \mapsto 10, y \mapsto 15\}$  and  $t_2 = t\sigma_2$  for  $\sigma_2 = \{x \mapsto 3.5, y \mapsto 20.2\}$ . Another generalization is  $t' = u + v$ .  $t_1$  can be obtained from  $t'$  by  $\sigma'_1 = \{u \mapsto 10, v \mapsto (15 \cdot 10)\}$ , and  $t_2$  by  $\sigma'_2 = \{u \mapsto 3.5, v \mapsto (20.2 \cdot 3.5)\}$ .<sup>1</sup> This generalization  $t'$  is not as special as  $t$ , it “forgets”, that the second operand of the addition is the product of two numbers where the second number is equal to the first operand.

Up to now we described *first-order* AU, which corresponds to cognitive models of analogy (Falkenhainer et al., 1989) insofar, as only objects can be mapped but relations must be preserved. First-order AU can be generalized to second-order, allowing that functions (or operators or relations) are generalized to function variables. Again we give a simple example:

$$t_3 = 10 + (15 \cdot 10) \text{ and } t_4 = 3.5 - (20.2 \cdot 3.5).$$

In the first-order case,  $t_3$  and  $t_4$  generalize to the object variable  $x$  because

---

<sup>1</sup>The usual definition of substitution is that variables can be replaced by arbitrary terms, that is, by other variables (“renaming”), by constants, or by more complex expressions.

**Fac-Unfolding:**

if  $n=0$  then 1  
 else if  $n=1$  then  $1 \cdot 1$   
 else if  $n=2$  then  $2 \cdot (1 \cdot 1)$   
 else if  $n=3$  then  $3 \cdot (2 \cdot (1 \cdot 1))$

**NSum-Problem:**

if  $n=0$  then 0  
 else if  $n=1$  then  $0-1$   
 else if  $n=2$  then  $(0-1)-2$   
 else if  $n=3$  then  $((0-1)-2)-3$

Figure 5: Example computations to be generalised

the expressions are built over different operators ( $(+)$  in  $t_3$  and  $(-)$  in  $t_4$ ). In the second-order case we obtain  $t_{3,4} = x F (y \cdot x)$  where  $x$  and  $y$  are object variables and  $F$  is a function variable with  $\sigma_3 = \{F \mapsto (+), x \mapsto 10, y \mapsto 15\}$  and  $\sigma_4 = \{F \mapsto (-), x \mapsto 3.5, y \mapsto 20.2\}$ . Because second-order AU allows for deletion, insertion and permutation of arguments of function variables, anti-instances and substitutions are written slightly differently (Hasker, 1995):  $t_{3,4} = F(x, \cdot(y, x))$  with  $\sigma_3 = \{F \mapsto +(\pi_1, \pi_2), x \mapsto 10, y \mapsto 15\}$  and  $\sigma_4 = \{F \mapsto -(\pi_1, \pi_2), x \mapsto 3.5, y \mapsto 20.2\}$ . All functions and operators are written in prefix notation. Function variables are represented with their arguments. Projections  $\pi$  describe which argument of the anti-instance is put at which position. For example,  $\pi_1$  returns the first argument, vic.  $x$ , of  $F(x, \cdot(y, x))$ .

We will illustrate the application of AU to programming by analogy with the factorial function as base problem:

**Fac-Problem:** If the factorial of 3 is calculated as  $3 \cdot 2 \cdot 1 \cdot 1$  what is the factorial for a natural number  $n$ ?

**Fac-Solution:**  $fac(n) = if(n=0, 1, n \cdot fac(n-1))$ .

If we unfold the recursive function of  $fac$ , that is, replace the recursive call in the body by the recursive definition, we obtain a computation as shown on the left-hand side of Fig. 5. An isomorphical problem is to calculate the *sum* of a natural number (Anderson & Thompson, 1989). Since we want to demonstrate that *second order AU* allows adaptation of non-isomorphical structures in a natural way, we use the *nsum* problem as a target problem:

**NSum-Problem:** If the neg. sum of 3 is calculated as  $((0-1)-2)-3$  what is the neg. sum for a natural number  $n$ ?

The problem can be enriched with an explanation of how the negative sum is calculated for numbers smaller than 3 (see right-hand side of Fig. 5). If we compare the unfolding of  $fac$  with the initial solution of  $nsum$  we see,

that the problems are structurally similar but not isomorphic. While *fac* associates to the right, *nsum* associates to the left.

Our second-order AU algorithm returns the anti-instance

**Fac-NSum-Generalization:**

```

if      n=0 then x
else if n=1 then 1 F x
else if n=2 then 2 F (1 F x)
else if n=3 then 3 F (2 F (1 F x))

```

with the substitutions  $\sigma_{\text{fac}} = \{x \mapsto 1, F \mapsto (\cdot \pi_1, \pi_2)\}$  and  $\sigma_{\text{nsum}} = \{x \mapsto 0, F \mapsto (-\pi_2, \pi_1)\}$ . Note that for  $\sigma_{\text{nsum}}$  the arguments of the subtraction operation are *reversed*. The abstract term captures the role of 1 and 0 as neutral element and of  $\cdot$  and  $-$  as ‘combination-operator’ respectively. The target solution, that is, the recursive program for calculating the negative sum can be obtained by applying the found substitutions to the recursive solution of the base problem *fac*:  $nsum(n) = \text{if}(n=0, 0, nsum(n-1) - n)$ .

Besides changing order of sub-terms, second-order AU allows for deletion and insertion of sub-terms (Hasker, 1995). In general, unrestricted second-order AU leads to infinite sets of most special generalizations. Therefore, our approach is restricted to “relevant combinators” allowing duplication but no deletion of symbols (Hasker, 1995). Even for restricted second-order AU, there no longer exists a unique most special generalization but a (finite) set of such generalizations. Additional heuristic criteria can be used to select a “most suitable” anti-instance from the set returned by the generic algorithm. Details and examples are given in Wagner (2002).

Second-order AU goes beyond Anderson’s cognitive model of programming by analogy (Anderson & Thompson, 1989) in several aspects: While in Anderson’s approach quite an extensive amount knowledge about the “function” and “form” of Lisp functions must be presented to the system in form of a hierarchy of schemes, we only need some rewrite rules to represent the problem as a term. Furthermore, Anderson’s approach only works for isomorphic problems, and, finally, generalization is dealt with as a separate process while in our approach it is obtained during analogical problem solving.

Finally, Anderson’s approach only addresses the problem of mapping and transfer. Generalization as last step of analogy is handled by a separate mechanism. In our approach, generalization is an integral part of the analogy process. Retrieval can be realized by ordering the anti-instances of all (or a pre-selection of) RPSs in memory with the target problem in a subsumption hierarchy and selecting the most specific anti-instance (Plaza, 1995; Schmid, Sinha, & Wysotzki, 2001).

## 6 Discussion

We have proposed to look at skill acquisition from the viewpoint of program synthesis. Thereby we obtain a unifying view for learning by doing and learning by analogy and have a sound theoretical background in computer science.

The crucial aspect of our approach is to realize a technique for the induction of problem schemes. It is a common assumption that people acquire knowledge about the structure of problems during problem solving experience for which exists some evidence in experimental cognitive psychology. But cognitive models as ACT-R or SOAR are neglecting to give methods for scheme induction. Inductive program synthesis is not explicitly a cognitive approach to fulfill this aim. But it realizes the ability of cognitive systems to detect regularities and generalize over them as described for example by Holland, Holyoak, Nisbett, and Thagard (1986).

Furthermore, the identification of data types from plans captures the evolution of perceptual chunks from problem solving experience (Koedinger & Anderson, 1990). For the simple *clearblock* example, the introduction of *topof(x)* in the program scheme represents the knowledge that the relevant part of the problem is the block lying on top of the currently focussed block. For more complex domains, as Tower of Hanoi, the data type represents “partial solutions” (see Schmid, 2003 for a detailed description).

The representation of problem schemes by recursive program schemes also differs from the representation formats proposed in cognitive psychology (Rumelhart & Norman, 1981). But RPSs are capturing exactly the characteristics which are attributed to cognitive schemes, namely that schemes represent procedural knowledge (“knowledge how”) which the system can interrogate to produce “knowledge that”, i. e., knowledge about the structure of a problem. Thereby problem schemes are suitable for modeling analogical reasoning.

Finally, RPSs have a clear formal definition. Therefore it is possible to define structural similarity between problem schemes in an unambiguous way. Using RPSs as representation format for source and target problems in analogical reasoning gives us a tool to construct problems which vary in their structural similarity in a systematic way. This makes it possible to perform experiments on the influence of structural similarity on successful transfer on a more detailed level (Schmid, Wirth, & Polkehn, 2003).

## References

- 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: Harvard University Press.
- Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science*, 13, 467-505.
- Anderson, J. R., & Lebière, C. (1998). *The atomic components of thought*. Mahwah, NJ: Erlbaum.
- Anzai, Y., & Simon, H. (1979). The theory of learning by doing. *Psychological Review*, 86, 124-140.
- Blum, A., & Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2), 281-300.
- Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1-2), 5-33.
- Carbonell, J. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine Learning - An Artificial Intelligence Approach* (Vol. 2, p. 371-392). Los Altos, CA: Morgan Kaufmann.
- Chomsky, N. (1959). Review of Skinner's 'Verbal Behavior'. *Language*, 35, 26-58.
- 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.
- Hasker, R. W. (1995). *The replay of program derivations*. Unpublished doctoral dissertation, Univ. of Illinois at Urbana-Champaign.
- Holland, J., Holyoak, K., Nisbett, R., & Thagard, P. (1986). *Induction - Processes of inference, learning, and discovery*. Cambridge, MA: MIT Press.
- Hummel, J., & Holyoak, K. (1997). Distributed representation of structure: A theory of analogical access and mapping. *Psychological Review*, 104(3), 427-466.
- Kitzelmann, E., Schmid, U., Mühlpfordt, M., & Wysotzki, F. (2002). Inductive synthesis of functional programs. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, & V. Sorge (Eds.), *Artificial intelligence, automated reasoning, and symbolic computation (AISC'02)* (p. 26-37). Springer, LNAI 2385.
- Koedinger, K., & Anderson, J. (1990). Abstract planning and perceptual chunks: Elements of expertise in geometry. *Cognitive Science*, 14, 511-550.
- Manna, Z., & Waldinger, R. (1987). How to clear a block: a theory of plans. *Journal of Automated Reasoning*, 3(4), 343-378.
- Martín, M., & Geffner, H. (2000). Learning generalized policies in planning using concept languages. In *Proc. 7th International Conference on Knowledge*

- Representation and Reasoning (KR 2000)* (p. 667-677). Morgan Kaufmann.
- Minton, S. (1985). Selectively generalizing plans for problem-solving. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI-85)* (pp. 596-599). Morgan Kaufmann.
- Mitchell, T. M. (1997). *Machine learning*. McGraw-Hill.
- Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.
- Newell, A., & Simon, H. A. (1972). *Human problem solving*. Englewood Cliffs, NJ: Prentice Hall.
- 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.
- Plaza, E. (1995). Cases as terms: A feature term approach to the structured representation of cases. In *Proc. 1st International Conference on Case-Based Reasoning (ICCBR-95)* (Vol. 1010, pp. 265-276). Springer.
- Plotkin, G. D. (1969). A note on inductive generalization. In *Machine intelligence* (Vol. 5, pp. 153-163). Edinburgh University Press.
- Reynolds, J. C. (1970). Transformational systems and the algebraic structure of atomic formulas. In *Machine intelligence* (Vol. 5, pp. 135-151). Edinburgh University Press.
- 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. 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.
- Schmid, U. (2003). *Inductive synthesis of functional programs - Learning domain-specific control rules and abstract schemes*. Heidelberg: Springer.
- Schmid, U., Burghardt, J., & Wagner, U. (2003). Anti-unification as an approach to analogical reasoning and generalization. In *Fifth International Conference on Cognitive Modeling, ICCM'03*. April 10 - 12, 2003, Bamberg, Germany.
- Schmid, U., & Kaup, B. (1995). Analoges Lernen beim rekursiven Programmieren (Analogical learning in recursive programming). *Kognitionswissenschaft*, *5*, 31-41.
- Schmid, U., Sinha, U., & Wysotzki, F. (2001). Program reuse and abstraction by anti-unification. In *Professionelles Wissensmanagement - Erfahrungen und Visionen* (p. 183-185). Shaker. (Long Version: <http://ki.cs.tu-berlin.de/~schmid/pub-ps/pba-wm01-3a.ps>)
- Schmid, U., Wirth, J., & Polkehn, K. (2003). A closer look on structural similarity

- in analogical transfer. *Cognitive Science Quarterly*, 3(1), 57-89.
- Schmid, U., & Wysotzki, F. (1998). Induction of recursive program schemes. In *Proc. 10th European Conference on Machine Learning (ECML-98)* (Vol. 1398, p. 214-225). Springer.
- Schmid, U., & Wysotzki, F. (2000). Applying inductive programm synthesis to macro learning. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)* (p. 371-378). AAAI Press.
- Schoppers, M. (1987). Universal plans for reactive robots in unpredictable environments. In *IJCAI '87* (p. 1039-1046). Morgan Kaufmann.
- 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 *Proc. 11th International Joint Conference on Artificial Intelligence (IJCAI-89), Detroit, MI*. Morgan Kaufman.
- Veloso, M. (1994). *Planning and learning by analogical reasoning*. Springer.
- Veloso, M., Carbonell, J., Pérez, M. A., Borrajo, D., Fink, E., & Blythe, J. (1995). Integrating planning and learning: The Prodigy architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), 81-120.
- Veloso, M. M., & Carbonell, J. G. (1993). Derivational analogy in Prodigy: Automating case acquisition, storage, and utilization. *Machine Learning*, 10, 249-278.
- Wagner, U. (2002). *Combinatorically restricted higher order anti-unification – An application to programming by analogy*. Unpublished master's thesis, Dept. of Electrical Engineering and Computer Science, TU Berlin, Germany. (<http://user.cs.tu-berlin.de/~xlat/>)

