Inductive Rule Learning on the Knowledge Level [1]

Ute Schmid [a] Emanuel Kitzelmann [b]

[a] *Faculty Information Systems and Applied Computer Science University of Bamberg, 96045 Bamberg, Germany*

[b] *International Computer Science Institute (ICSI) Berkeley, CA, USA*

Abstract

We present an application of the analytical inductive programming system IGOR to learning sets of recursive rules from positive experience. We propose that this approach can be used within cognitive architectures to model regularity detection and generalization learning. Induced recursive rule sets represent the knowledge which can produce systematic and productive behavior in complex situations – that is, control knowledge for chaining actions in different, but structural similar situations. We argue, that an analytical approach which is governed by regularity detection in example experience is more plausible than generate-and-test approaches. After introducing analytical inductive programming with IGOR we will give a variety of example applications from different problem solving domains. Furthermore, we demonstrate that the same generalization mechanism can be applied to rule acquisition for reasoning and natural language processing.

1 Introduction

The human ability to master complex demands is to a large extend based on the ability to exploit previous experiences. Based on our experience, we are

Email addresses: `ute.schmid@uni-bamberg.de` (Ute Schmid), `emanuel@icsi.berkeley.edu` (Emanuel Kitzelmann).

[1] A considerable shorter version of this paper was presented at the AGI'09 as Schmid, Ute, Hofmann, Martin, and Kitzelmann, Emanuel (2009). Analytical Inductive Programming as a Cognitive Rule Acquisition Device. In: Ben Goertzel, Pascal Hitzler, and Marcus Hutter (Eds.), *Proceedings of the Second Conference on Artificial General Intelligence* (pp. 162–167). Amsterdam: Atlantis Press.

able to predict characteristics or reactions of (natural or man-made, inanimate or animate) objects, we can reason about possible outcomes of actions, and we can apply previously successful routines and strategies to new tasks and problems. In philosophy, psychology and artificial intelligence, researchers proposed that the core process to expand knowledge, that is, construct hypotheses, in such a way that we can transfer knowledge from previous experience to new situations is *inductive* inference (Goodman, 1965; Klahr & Wallace, 1976; Holland, Holyoak, Nisbett, & Thagard, 1986; Tenenbaum, Griffiths, & Kemp, 2006).

From the perspective of machine learning, inductive inference is characterized as a mechanism for generalizing over observed regularities in examples. To allow for generalization, every learning algorithm must make some *a priori* assumptions. This so called inductive bias gives the rational basis to allow transfer of the learned hypothesis to new situations (Mitchell, 1997). The *restriction* or language bias characterizes the language in which induced hypotheses are represented. The *preference* or search bias characterizes the mechanism for selecting hypothesis. Every machine learning algorithm, be it statistical, neuronal or symbolical, makes such (maybe implicit) assumptions. This basic proposition of machine learning certainly also holds for human induction. That is, human learning is very powerful but nevertheless learning is restricted by the way knowledge can be represented and by some mechanism for preferring some generalizations over others. The human preference mechanism is a very clever one (Holland et al., 1986): it appears to draw from a vast amount of background knowledge and is highly flexible – allowing to perform eager generalizations in one context and being more conservative in another context.

Human learning is considered a base mechanism which takes place on all levels from sensu-motor associations over pattern recognition and concept acquisition to learning high-level schemata, rules and strategies on the knowledge level (Sun, 2007). It can be safely assumed that different mechanisms and biases are at work on these different levels. In this paper we focus on learning of complex structures and routines on the knowledge level. More precisely, we want to demonstrate that an approach to inductive programming can be applied to model the acquisition of generalized rules from example experience. We specifically are interested in rules which are *productive* (Chomsky, 1965) in such a sense that they can be applied in situations of various complexity – for example, rules which can characterize the ability of humans to grasp the transitivity of some concepts such as *ancestor*. Typically, we get acquainted with such a concept by some, necessarily small and finite examples, such as *my father is my ancestor*, *my grandfather is my ancestor*, and my *grand-grand-father* is also my ancestor and are able to induce the infinite regularity of the concept which allows us to apply it to ancestor relations of various complexities. Likewise, productive rules govern the generation and application of regular action sequences. For example, if humans have learned how to solve

Tower of Hanoi problems with three and four discs, at least some of them are able to generalize the underlying strategy for solving problems with an arbitrary number of discs (Kotovsky, Hayes, & Simon, 1985; Anderson & Douglass, 2001; Altmann & Trafton, 2002; Anderson, Albert, & Fincham, 2005; Welsh & Huizinga, 2005).

Acquired concepts and strategic rules often can be communicated, that is, they must be available as declarative structures on the knowledge level (Newell, 1982; Anderson, 1983). For that reason, we are interested in inductive programming as a *symbolic* approach to learning. While statistical and neuronal approaches present powerful mechanisms for inductive learning, the acquired (black box) classifiers typically can not or not easily be transformed into (white box) symbolic representations (Hammer & Hitzler, 2007) which would be necessary for incorporating the acquired knowledge in a representation on the knowledge level.

In the following we shortly introduce our analytical inductive programming system Igor. Then we present the general setting for incorporating Igor in a cognitive architecture. Afterwards we will give various examples of cognitive rule acquisition in the domain of problem solving. We will shortly demonstrate that the same mechanisms can be applied to learning in the context of reasoning and language understanding and we will conclude with a discussion and further work to be done.

2 Analytical Inductive Programming with Igor

Inductive programming research addresses the problem of learning computer programs from incomplete specifications, typically samples of the desired input/output behavior and possibly additional constraints (Biermann, Guiho, & Kodratoff, 1984; Flener & Schmid, 2009; Schmid, Kitzelmann, & Plameijer, 2010). Induced programs are usually represented in a declarative – functional or logic – programming language. As a special application of machine learning (Mitchell, 1997), inductive programming creates program *hypotheses*, that is, generalized, typically recursive, programs. In contrast to classification learning, program hypotheses must cover *all* given examples correctly since for programs it is expected that a desired input/output relation holds for all possible inputs.

There are two general approaches to inductive programming: analytical and generate-and-test methods. Generate-and-test methods enumerate syntactically correct programs and test these programs against the given examples guided by some search strategy. For example, the inductive logic programming (ILP) system Foil constructs sets of Horn clauses by sequential covering, that is by a first construct then test approach. This approach was also applied to learning recursive rule sets (Quinlan & Cameron-Jones, 1995). The most successful search-based system is Adate (Olsson, 1995) which constructs ML programs using evolutionary principles. The system MagicHaskeller (Katayama, 2005) enumerates Haskell programs with higher-order functions.

Clearly, it is not plausible to assume that a cognitive system learns rules by a generate-and-test approach. In contrast, in analytical methods learning is guided by the structure underlying the given examples. Early research on inductive programming in the nineteen-seventies brought forward such analytical strategies for learning Lisp programs from small sets of positive input/output examples (Biermann et al., 1984). The most influential of the early systems is Thesys (Summers, 1977). It realized a two step approach to synthesize programs: In a first step, input/output examples were rewritten into traces, in a second step recurrent patterns were searched-for in the traces and the found regularities were generalized to a recursive function.

For Thesys and all later analytical approaches it is enough to present a small set of only positive input/output examples. These examples must be the first representants of the underlying data-type of the input parameter. In contrast, in generate-and-test approaches, an arbitrary set of positive examples can be presented. In addition negative examples can and often must be used to eliminate unsuitable hypotheses.

Igor2 (Kitzelmann, 2009, 2010) was developed as a successor to the classical Thesys system and its generalization Igor1 (Kitzelmann & Schmid, 2006). To our knowledge, Igor2 is currently the most powerful system for analytical inductive programming. Its scope of inducible programs and the time efficiency of the induction algorithm compares favorably with inductive logic programming and other approaches to inductive programming (Hofmann, Kitzelmann, & Schmid, 2008). The system is realized in the constructor term rewriting system Maude. Therefore, all constructors specified for the data types used in the given examples are available for program construction. Since Igor2 is primarily designed as an assistant system for *program* induction, it relies on small sets of noise-free positive input/output examples and it cannot deal with uncertainty. Furthermore, the examples have to be the first inputs with respect to the complexity of the underlying data type. Given these restrictions, Igor2 can guarantee that the induced program covers *all* examples correctly and provides a minimal generalization over them. Classification learning for noise-free examples such as `PlayTennis` (Mitchell, 1997) can be performed as a special case (Kitzelmann, 2009).

Igor2 specifications consist of a set of examples as described above together with a specification of the input data type. Background knowledge for additional functions can (but needs not) be provided. IGOR2 can induce several dependent target functions (i.e., mutual recursion) in one run. Auxiliary functions are invented if needed. In general, a set of rules is constructed by generalization of the input data by introducing patterns and predicates to partition the given examples and synthesis of expressions computing the specified outputs. Partitioning and search for expressions is done systematically and completely which is tractable even for relative complex examples because construction of hypotheses is data-driven. IGOR2's restriction bias is the set of all functional recursive programs where the outermost function must be either non-recursive or provided as background knowledge.

IGOR2's built-in preference bias is to prefer fewer case distinctions, most specific patterns and fewer recursive calls. Thus, the initial hypothesis is a single rule per target function which is the least general generalization (Plotkin, 1969) of the example equations. If a rule contains unbound variables on its right-hand side, successor hypotheses are computed using the following operations: (i) Partitioning of the inputs by replacing one pattern by a set of disjoint more specific patterns or by introducing a predicate to the right-hand side of the rule; (ii) replacing the right-hand side of a rule by a (recursive) call to a defined function where finding the argument of the function call is treated as a new induction problem, that is, an auxiliary function is invented; (iii) replacing sub-terms in the right-hand side of a rule which contain unbound variables by a call to new subprograms.

3 IGOR as Cognitive Rule Acquisition Device

In the context of a cognitive architecture IGOR can be seen as a module which observes the content of the working memory and greedily tries to generalize over its content. In many cases, IGOR will not detect regularities. This can be because for the current cognitive task there exists no underlying regularity or because the current problem representation is structured in such a way that the regularities cannot be detected. In the second case, one can hope that over several problem solving trials in the domain eventually the problem is represented such that regularity detection becomes possible.

While we are aware that finding the "right" representation is a crucial problem (Kaplan & Simon, 1990), in this paper we focus on IGOR as a powerful and fast mechanism for constructing minimal generalizations for complex domains. In the domain of problem solving, we explored some strategies for rewriting action sequences into suitable representations for IGOR (Schmid & Wysotzki, 2000; Kitzelmann, 2003; Schmid, 2003).

If IGOR2 produces a generalization over examples, the induced rules are available as symbolic representations in the working memory. Following the usual assumption that active components of classical working memory are conscious (Baars & Franklin, 2003), this can be interpreted as some realization of the "aha"-experience – the sudden impression that one has understood a problem (Bühler, 1907; Ohlsson, 1984; Knoblich, Ohlsson, Haider, & Rhenius, 1999). The experience of having a sudden insight is well-known to every human problem solver. We all remember episodes where we suddenly "saw" the solution procedure for some problem and "knew" that this is the right one. More systematic empirical evidence of this phenomenon is for example available for Tower of Hanoi where it could be shown that successful problem solvers can verbalize the components of the recursive solution strategy (Welsh & Huizinga, 2005).

There are some aspects of Igor which might be considered as making this approach implausible as a cognitive mechanism. First, one might argue that humans notoriously have difficulties in understanding recursion (Kahney, 1989) and therefore that representing expertise by recursive rule sets is inadequate. We argue that such recursive rule sets are just a means to represent the competence of producing systematic and productive behavior in a given domain. Nobody would argue that neural nets are inadequate for representing human skills because humans are not able to multiply large matrices of real numbers on the fly. Furthermore, all approaches to classification learning which do *not* take into account recursion – be it symbolic approaches or sub-symbolic or statistical approaches – will always be bound by some fixed size of the input to produce a meaningful output. For example, such approaches might correctly learn an ancestor-relation up to the tenth grandfather but be unable to produce a useful output for the eleventh one; they would produce correct action sequences for solving Tower of Hanoi problems up to five discs but not for more discs, and so on.

Second, one might argue that such a brittle approach as analytical inductive programming does not account for human errors since, once the correct rule set is induced, the system will always generate correct solutions. Here we argue that Igor models learning on the *competence* level. We do not dispute the fact that there are many factors which influence *performance* (Chomsky, 1965). There are many reasons why a person who has acquired a correct set of rules might produce errors when solving a given problem in a given context. Typical sources of error are working memory restrictions or motivational or emotional states which influence information processing (Bach, 2009; Simon, 1958).

Third, one might argue that a mechanism which can only induce correct rules if the first examples with respect to some underlying data type are presented and that this examples are error-free is not suitable for modeling learning in complex domains. Here we have a trade-off between analytical approaches where rule construction is guided by the examples and generate-and-test approaches. Generate-and-test approaches are robust with respect to variance in complexity of the given examples and with respect to erroneous examples. However, there is a price to pay: in such approaches, huge spaces of possible rule sets are generated and tested against the given examples. It is not plausible to assume that such kind of search processes take place in working memory. Furthermore, these approaches typically are much slower than IGOR2 which typically can induce a rule set in milliseconds (Hofmann et al., 2008). In general, there will be a trade-off between the amount of search (and search space) necessary to produce generalized rule sets and the requirements which constrain the examples. Fortunately, in many domains, obtaining the k first positive examples comes quite naturally. As we will show in the following section, in problem solving typically the examples are derived from the problem solving traces which can be ordered by the number of actions which were performed.

4 Rule Induction in Problem Solving

In cognitive psychology speed-up effects in problem solving are often modelled as composition of primitive rules as a result of their co-occurrence during problem solving, e.g., knowledge compilation in ACT (Anderson & Lebière, 1998) or operator chunking in SOAR (Rosenbloom & Newell, 1986). Similarly, in AI planning macro learning was modelled as composition of primitive operators to more complex ones (Minton, 1985; Korf, 1985). We will not dispute that performance improvements in problem solving can often be explained by such mechanisms for a more efficient organization of the rule base. But, there is empirical evidence that humans are able to acquire general problem solving strategies from problem solving experiences, that is, that generalized strategies are learned from sample solutions. For example, after solving Tower of Hanoi problems, at least some people have acquired the recursive solution strategy (Anzai & Simon, 1979; Welsh & Huizinga, 2005). Typically, experts are found to have superior *strategic* knowledge in contrast to novices in a domain (Meyer, 1992).

There were some proposals to the learning of domain specific control knowledge in AI planning (Shell & Carbonell, 1989; Shavlik, 1990; Martín & Geffner, 2000). All these approaches proposed to learn cyclic/recursive control rules which reduce search. Learning recursive control rules, however, will eliminate search completely. With enough problem solving experience, some generalized strategy, represented by a set of rules (equivalent to a problem solving *scheme*) should be induced which allows a domain expert to solve this problem via application of his/her strategic knowledge. We already tried out this idea using IGOR1 (Schmid & Wysotzki, 2000). However, since IGOR1 was a two-step approach where examples had to be first rewritten into traces and afterwards recurrence detection was performed in these traces, this approach was restricted in its applicability. With IGOR2 we can reproduce the results of IGOR1 faster and without specific assumptions to preprocessing and furthermore can tackle more complex problem domains.[2]

The general idea of learning domain specific problem solving strategies is that first some small sample problems are solved by means of some planning or problem solving algorithm and that then a set of generalized rules are learned from this sample experience. This set of rules represents the competence to solve arbitrary problems in this domain.

4.1 How to Clear a Block

We illustrate the idea of our approach with the simple *clearblock* problem (see Figure 1). Inference of the recursive strategy for this problem was also researched in the context of deductive program synthesis (Manna & Waldinger, 1987). A problem consists of a set of blocks which are stacked in some arbitrary order. The problem solving goal is that one specific block – in our case A – should be cleared such that no block is standing above it. We use predicates *clear(x)*, *on(x, y)*, and *ontable(x)* to represent problem states and goals. The only available operator is *puttable*: A block x can be put on the table if it is clear (no block is standing on it) and if it is not already on the table but on another block. Application of *puttable(x)* has the effect that block x is on the table and the side-effect that block y gets cleared if *on(x, y)* held before operator application. The negative effect is that x is no longer on y after application of *puttable*.

We use a PDDL-like[3] notation for the problem domain and the problem descriptions. The problem domain is given by the *puttable* operator. We defined four different problems of small size each with the same problem solving goal (*clear(A)*) but with different initial states: The most simple problem is the case where A is already clear. This problem is presented in two variants – A is on the table and A is on another block – to allow the induction of a *clearblock* rule for a block which is positioned in an arbitrary place in a stack. The third initial state is that A is covered by one block, the fourth that A is covered by two blocks. A planner might be presented with the problem domain and problem descriptions given in Figure 1.

The resulting action sequences can be obtained by any PDDL planner (Ghal-

[2] The complete data sets and results for all problems discussed can be found on www.cogsys.wiai.uni-bamberg.de/effalip/download.html.
[3] see `http://ls5-www.cs.tu-dortmund.de/~edelkamp/ipc-4/pddl.html`

Problem domain:
```
puttable(x)
PRE: clear(x), on(x, y)
EFFECT: ontable(x), clear(y), not on(x,y)
```
Problem Descriptions:
```
: init-1 clear(A), ontable(A)
: init-2 clear(A), on(A, B), ontable(B)
: init-3 on(B, A), clear(B), ontable(A)
: init-4 on(C, B), on(B, A), clear(C), ontable(A)
: goal clear(a)
```
Problem Solving Traces/Input to Igor2
```
fmod CLEARBLOCK is
  *** data types, constructors
  sorts Block Tower State .
  op table : -> Tower [ctor] .
  op __ : Block Tower -> Tower [ctor] .
  op puttable : Block State -> State [ctor] .
  *** target function declaration
  op ClearBlock : Block Tower State -> State [metadata "induce"] .
  *** variable declaration
  vars A B C : Block .
  var S : State .
  *** examples
  eq ClearBlock(A, A table, S) = S .
  eq ClearBlock(A, A B table, S) = S .
  eq ClearBlock(A, B A table, S) = puttable(B, S) .
  eq ClearBlock(A, C B A table, S) = puttable(B, puttable(C, S)) .
endfm
```
Induced Clearblock Strategy (4 examples, 0.036 sec)
```
ClearBlock(A, (B T), S) = S  if A == B
ClearBlock(A, (B T), S) =
  ClearBlock(A, T, puttable(B, S))  if A =/= B
```

Fig. 1. Initial experience with the *clearblock* problem with three blocks and solution

lab, Nau, & Traverso, 2004) and rewritten to Igor2 (i.e. Maude) syntax. When rewriting plans to Maude equations (see Figure 1) we give the goal, that is, the name of the block which is to be cleared, as first argument. The second argument represents the initial state, that is, the stack as list of blocks and *table* as bottom block. The third argument is a situation variable (McCarthy, 1963; Green, 1969; Manna & Waldinger, 1987; Schmid & Wysotzki, 2000) representing the current state. Thereby plans can be interpreted as nested function applications and plan execution can be performed on the content of the situation variable. The right-hand sides of the example equations correspond to the action sequences which were constructed by a planner, rewritten as nested terms with situation variable S as second argument of the *puttable* operator. Currently, the transformation of plans to examples for Igor2 is done “by hand”. For a fully automated interface from planning to inductive programming, a set of rewrite rules must be defined.

Given the action sequences for clearing a block up to three blocks deep in a stack as initial experience, Igor2 generalizes a simple tail recursive rule system which represents the competence to clear a block which is situated in arbitrary depth in a stack (see Figure 1). That is, from now on, it is no longer necessary to search for a suitable action sequence to reach the *clearblock* goal. Instead, the generalized knowledge can be applied to produce the correct action sequence directly. Note, that Igor2 automatically introduced the equal

Problem Solving Traces/Input to IGOR2
```
fmod ROCKET is
  *** data types, constructors
  sorts Object OList State InVec .
  op nil : -> OList [ctor] .
  op __ : Object OList -> OList [ctor] .
  ops load unload : Object State -> State [ctor] .
  op move : State -> State [ctor] .
  *** target function declaration
  op Rocket : OList State -> State [metadata "induce"] .
  *** variables
  vars O1 O2 : Object .
  var S : State .
  *** examples
  eq Rocket(nil, S) = move(S) .
  eq Rocket((O1 nil), S) = unload(O1, move(load(O1, S))) .
  eq Rocket((O1 O2 nil), S) = unload(O1, unload(O2, move(load(O2, load(O1, S))))) .
endfm
```
Rocket (3 examples, 0.012 sec)
```
Rocket(nil, S) = move(S) .
Rocket((O Os), S) = unload(O, Rocket(Os, load(O, S)))
```

Fig. 2. Example equations and learned rules for the *rocket* domain

predicate to discern cases where A is on top of the stack from cases where A is situated farther below since these cases could not be discriminated by disjoint patterns on the left-hand sides of the rules.

4.2 Rocket Transport

A more complex problem domain is *rocket* (Veloso & Carbonell, 1993). This domain was originally proposed to demonstrate the need of interleaving goals. The problem is to transport a number of objects from earth to moon where the rocket can only fly in one direction. That is, the problem cannot be solved by first solving the goal *at(o1, moon)* by loading it, moving it to the moon and then unloading it. Because with this strategy there is no possibility to transport further objects from earth to moon. The correct procedure is first to load all objects, then to fly to the moon and finally to unload the objects. IGOR2 learned this strategy from examples for zero to two objects (see Figure 2). The objects to be loaded and unloaded are provided as a list (sort `OList` with constructors `nil` for the empty list and the "empty syntax" constructor `__` to construct a list from an object and a list).

Let us demonstrate—by means of the *rocket* problem—how analytical inductive programming systems and especially IGOR2 derive a recursive generalization from example equations.

The three provided *rocket* examples (cp. Figure 2) are:

```
1. Rocket(nil, S) = move(S)
2. Rocket((O1 nil), S) = unload(O1, move(load(O1, S)))
3. Rocket((O1 O2 nil), S) = unload(O1, unload(O2, move(load(O2, load(O1, S)))))
```

As an initial hypothesis, IGOR2 takes the least general generalization (LGG) over all example equations. That means, the three equations are scanned in parallel from left to right, position by position. If the symbol at a position is the same in all example equations, it is kept for that position. If the symbols differ, a variable is introduced at that position. For each introduced variable, the respective substitutions to recover the corresponding sub-terms in the examples are stored. If at one position a variable is to be introduced, it is checked if the resulting substitutions are already associated with an earlier introduced variable and if yes, that variable is taken. Consider for example the root (left-most) position of the left-hand sides (LHSs). The symbol is `Rocket` in all three examples, hence `Rocket` is kept as symbol at the root position of the LHS of the LGG. In contrast, consider the right-hand sides (RHSs) of the examples. They differ at the root position (`move` in the first equation, `unload` in the remaining two equations) such that the RHS of the LGG consists of a single variable.

The LGG, i.e., the initial hypothesis for the three *rocket* example equations above is

```
  Rocket(Os, S) = S'.
```

The variable Os results from the different constructors `nil` and `__` at the same position in the example equations. The variable S' results from the different constructors `move` and `unload` as roots of the RHSs in the examples.

This initial hypothesis is regarded *unfinished* due to the variable S' as RHS which does not occur in the LHS such that this initial hypothesis does not represent a function. We call such variables *open* variables. Now IGOR2, in principle, applies three refinement operators in parallel to this unfinished hypothesis: (i) It refines the pattern (LHS) by a set of more specific patterns, leading to a new hypothesis consisting of a *set* of equations; (ii) it considers open *sub*terms of the RHS as new subproblems; (iii) it replaces the open RHS by a (recursive) function call. All these operations are data-driven such that many potential refinements can be ruled out a-priori. In particular, for our initial *rocket* hypothesis, operators (ii) and (iii) do not apply: Since the RHS consists of the single, non-structured, variable S', there are no proper subproblems to deal with. Further, since no example RHS subsumes another one by some substitution, a hypothetical recursive call of *rocket* as replacement for S' can be ruled out.

It remains the first operator—introducing a case distinction by refining the pattern in the LHS. Therefore, IGOR2 checks which of the pattern variables in the initial hypothesis result from different constructors in the example equations. As mentioned above, in our example, the pattern variable Os results from the two different constructors `nil` and `__` in the first and in the second and third example equation, respectively. Now the example equations are partitioned according to these different constructors such that the first example equation goes into one subset and the second and third example into a second subset. The single equation of the initial hypothesis is then replaced by initial equations (LGGs) of the two subsets, leading to the refined hypothesis:

```
Rocket(nil, S) = move(S)
Rocket((O Os), S) = unload(O, S')
```

The second equation is again unfinished due to the variable S'. Since the open variable S' now occurs as a proper sub-term in the unfinished RHS, it can be dealt with as a subproblem. Therefore, IGOR2 replaces the S' by a call to a new sub-function `Sub`,

```
Rocket((O Os), S) = unload(O, Sub((O Os), S)),
```

and abduces examples for this new sub-function by simply taking the appropriate sub-terms of the RHSs of the corresponding *rocket* example equations:

```
Sub((O1 nil), S) = move(load(O1, S))
Sub((O1 O2 nil), S) = unload(O2, move(load(O2, load(O1, S))))
```

The refinement step is finished by computing an initial hypothesis for the abduced examples for `Sub` and adding it to the refined hypothesis for *rocket*, leading to the following result:

```
Rocket(nil, S) = move(S)
Rocket((O Os), S) = unload(O, Sub((O Os), S))
Sub((O Os), S) = S'
```

The equation for `Sub` remains open. At this step, the RHSs of both example equations for `Sub` are subsumed by an RHS of the example equations for *rocket*. In particular, the RHS

```
move(S)
```

of the first *rocket* example subsumes the RHS

```
move(load(O1, S))
```

of the first `Sub` example with the substitution $\{S \mapsto load(O1, S)\}$ and the RHS

```
unload(O1, move(load(O1, S)))
```

of the second *rocket* example subsumes the RHS

```
unload(O2, move(load(O2, load(O1, S))))
```

of the second Sub example by $\{O1 \mapsto O2, S \mapsto load(O1, S)\}$.

This indicates that the Sub examples might be computable by calling *rocket*. IGOR2 now applies the found substitutions to the respective *LHSs* of the *rocket* examples and then tries to generate a term over the pattern variables O, Os, S such that the LHSs of the Sub examples are mapped to the corresponding, substituted, LHSs of the *rocket* examples. The found term,

```
(Os, load(O, S))
```

is taken as the argument of the *rocket* call, leading to the final hypothesis:

```
Rocket(nil, S) = move(S)
Rocket((O Os), S) = unload(O, Sub((O Os), S))
Sub((O Os), S) = Rocket(Os, load(O, S))
```

Since the Sub function is not recursive, it can be eliminated by unfolding, leading to the solution stated in Figure 2.

4.3 Building Towers

A most challenging problem domain which is still used as a benchmark for planning algorithms is *blocks-world.* A typical blocks-world problem is to build a *tower* of some blocks in some prespecified order. With evolutionary programming, an iterative solution procedure to this problem was found from 166 examples (Koza, 1992). The found strategy was to first put all blocks on the table and than build the tower. This strategy is clearly not efficient and cognitively not very plausible. If, for example, the goal is a tower *on(A, B), on(B, C)* and the current state is *on(C, B), on(B,A)*, even a young child will first put C on the table and then *directly* put B on C and not put B on the table first. Another proposal to tackle this problem is to learn decision rules which at least in some situations can guide a planner to select the most suitable action (Martín & Geffner, 2000). With the learned rules, 95.5% of 1000 test problems were solved for 5-block problems and 72.2% of 500 test problems were solved for 20-block problems. The generated plans, however, are about two steps longer than the optimal plans. In Figure 3 we present the rules IGOR2 generated from only nine example solutions. This rule system will always produce the optimal action sequence.

To illustrate how examples were presented to IGOR2 we show one example in Figure 4. The goal is to construct a tower for some predefined ordering of blocks. To represent this ordering, blocks are represented constructively as

Tower (9 examples of towers with up to four blocks, 1.2 sec)
(additionally: 10 corresponding examples for Clear and IsTower predicate as background knowledge)
```
Tower(O, S) = S  if IsTower(O, S)
Tower(O, S) =
  put(O, Sub1(O, S),
    Clear(O, Clear(Sub1(O, S),
      Tower(Sub1(O, S), S))))  if not(IsTower(O, S))
Sub1(s(O), S) = O .
```

Fig. 3. Induced optimal strategy for solving arbitrary *tower* problems

```
eq Tower(s s table,
         ((s s s s table) (s table) table | ,
          (s s s table) (s s table) table | , nil)) =
  put(s s table, s table,
      put(s s s table, table,
          put(s s s s table, table,
              ((s s s s table) (s table) table | ,
               (s s s table) (s s table) table | , nil)))) .
```

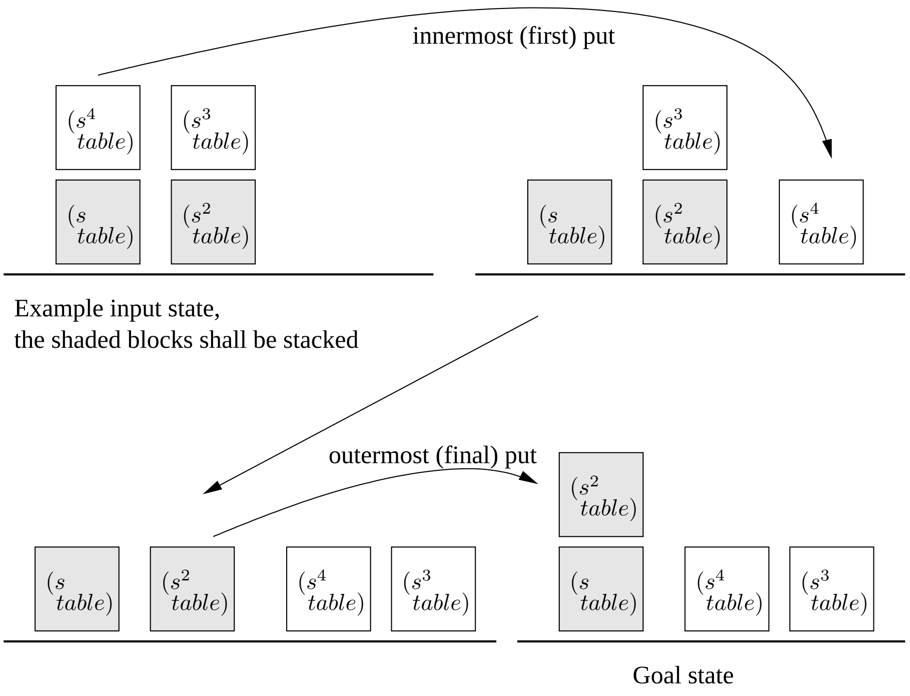

Fig. 4. One of the nine example equations for *tower*

"successors" to the table with respect to the goal state (| representing the empty tower). Therefore the top object of the tower to be constructed is given as first argument of the *tower* function. If the top object is *s s s table*, the goal is to construct a tower with three blocks with *s table* on the table, *s s table* on *s table* and *s s s table* on *s s table*. The second argument again is a situation variable which initially holds the initial state. In the example in Figure 4 *s s table* (we may call it block 2) shall be the top object and the initial state consists of two towers, namely block 4 on block 1 and block 3 on block 2. That is, the desired output is the plan to get the tower block 2 on block 1. Therefore blocks 1 and 2 have to be cleared, these are the both innermost puts, and finally block 2 has to be stacked on block 1 (block 1 lies on the table already), this is the out-most put.

In addition to the *tower* example, IGOR2 was given an auxiliary function *IsTower* as background knowledge. This predicate is true if the list of blocks presented to it are already in the desired order. Furthermore, we did not learn the *Clear* function used in *tower* but presented some examples as background knowledge.

4.4 Car Park Problems

In 2008, the biennial *international planning competition (IPC)* for the first time included a special *learning track*[4]. The learning track contains two phases: A learning phase, where the planners solve example problems in the competition domains; and an evaluation phase, where they solve further problems in the same domains and where they may use domain-specific control-knowledge—e.g., domain-specific heuristics, macro operators, or policies—that they learned based on solving the problems in the learning phase.

One of the used domains was the *parking domain*. Several cars are to be parked into N curb locations, where each curb location can carry up to two cars. If two cars are "stacked" into one location, only the back-most car can move. If there are N curbs, then there are $2 * (N - 1)$ distinguished cars; two parking positions are always free. A problem consists of re-parking all cars from a particular initial configuration to a particular target configuration.

The domain is isomorphic to a variant of the *blocksworld*: Instead of building towers anywhere at the table, there are N possible tower-locations and each tower can maximally contain two blocks. Furthermore, there are exactly $2 * (N - 1)$ involved blocks.

We consider a generalized version of this car parking domain—namely a generalization to M (instead of 2) cars that can be stacked into one curb location. If there are N locations, then there are $M * (N - 1)$ cars involved.

Coming up with an (optimal) solution strategy for this problem class is even harder than for tower building in the ordinary blocksworld, because there are more constrains regarding applicability of operators. A general strategy could be to fill one parking position after the other with the correct car (according to the goal configuration) where the parking positions are ordered from innermost to outermost in each curb location. (In the equivalent constraint blocksworld domain this would correspond to get the correct blocks position by position where positions in each tower are ordered from bottom to top.) However, filling

[4] Homepage of the IPC: `http://www.icaps-conference.org/index.php/Main/Competitions`, homepage of the 2008 learning track: `http://eecs.oregonstate.edu/ipc-learn/`.

```
Input to IGOR2
  eq PutLast(Car1, Curb, Cars, 0, S) = S .
  eq PutLast(Car1, Curb, Cars, s 0, S) = move(Car1, Curb, S) .
  eq PutLast(Car1, Curb, Car2 Cars, s s 0, S) =
       move(Car1, Curb, move(Car2, Curb, S)) .
  eq PutLast(Car1, Curb, Car2 Car3 Cars, s s s 0, S) =
       move(Car1, Curb, move(Car3, Curb, move(Car2, Curb, S))) .
Induced Rules (4 examples, 0.024 sec)
  PutLast(Car1, Curb, Cars, 0, S) = S
  PutLast(Car1, Curb, Cars, s 0, S) = move(Car1, Curb, S)
  PutLast(Car1, Curb, Car2 Cars, s s N, S) =
    PutLast(Car1, Curb, Cars, s N,
      move(Car2, Curb, S))
```

Fig. 5. Posing the described *parking* sub-problem for IGOR2 and induced recursive rule set

one particular position is itself not a trivial task. Obviously, it comprises (i) clearing the correct car such that no other car is stacked behind it and (ii) clearing the particular parking position such that the curb location including this position is only filled up to the preceding position. These two sufficient preconditions for moving the correct car to the chosen position imply an even stronger precondition. Consider the particular task of moving car A to the innermost position in some curb location. That means, the target curb location must be completely free, no car may already be parked in it. Yet since there are N locations and $M * (N-1)$ cars and each location only can take M cars, this implies that all other curb location are completely filled. Hence, our car A, in order to be movable, must be located at the outermost position of any curb location.

That is, one relevant subproblem of the complete parking problem class is to fill one particular curb location completely with a particular car at the outermost position. We have considered this particular subproblem. The function to be learned was `PutLast`. It takes five parameters: The selected car, the selected curb location, a list of further (possibly all remaining) cars that can be moved according to their order in the list, a number indicating the remaining number of free positions in the selected curb location, and a state. We assume, that the selected car to be moved to the out-most position is already cleared, hence movable. Figure 5 shows the four examples we provided to IGOR and the learned rules. The learned strategy is to take cars from the list of cars and move them into the selected curb location until only one free position remains. Then move the selected car into this outermost position.

4.5 Tower of Hanoi

Finally, the recursive solution to the Tower of Hanoi problem was generated by IGOR2 from three examples (see Figure 6).

For the discussed typical problem solving domains IGOR2 could infer the re-

Input to Igor2
```
  eq Hanoi(0, Src, Aux, Dst, S) =
    move(0, Src, Dst, S) .
  eq Hanoi(s 0, Src, Aux, Dst, S) =
    move(0, Aux, Dst,
      move(s 0, Src, Dst,
        move(0, Src, Aux, S))) .
  eq Hanoi(s s 0, Src, Aux, Dst, S) =
    move(0, Src, Dst,
      move(s 0, Aux, Dst,
        move(0, Aux, Src,
          move(s s 0, Src, Dst,
            move(0, Dst, Aux,
              move(s 0, Src, Aux,
                move(0, Src, Dst, S))))))) .
```
Induced Tower of Hanoi Rules (3 examples, 0.076 sec)
```
Hanoi(0, Src, Aux, Dst, S) = move(0, Src, Dst, S)
Hanoi(s D, Src, Aux, Dst, S) =
  Hanoi(D, Aux, Src, Dst,
    move(s D, Src, Dst,
      Hanoi(D, Src, Dst, Aux, S)))
```

Fig. 6. Posing the *Tower of Hanoi* problem for Igor2 and induced recursive rule set

cursive generalizations very fast and from small example sets. The learned recursive rule systems represent the strategic knowledge to solve all problems of the respective domains with a minimal number of actions. As it was to be expected, Tower of Hanoi can be learned faster and with less examples than *tower*. This is contrary to what we know from human cognitive development: Tower of Hanoi is the by far more intellectually complex problem. While already small children can build towers of blocks in some order, they usually are not able to produce the optimal action sequence for Tower of Hanoi. This difference between human intelligence and artificial intelligence is also reflected in planning which is faster for Tower of Hanoi than for *tower* problems due to the fact that the the problem space of Tower of Hanoi is smaller, highly symmetric and operator application is more constrained.

5 Rule Induction in Reasoning and Language

A classic work in the domain of reasoning is how humans induce rules in concept learning tasks (Bruner, Goodnow, & Austin, 1956). Indeed, this work has inspired the first decision tree algorithms (Hunt, Marin, & Stone, 1966). This work addressed simple conjunctive or more difficult to acquire disjunctive concepts. However, people are also able to acquire and correctly apply recursive concepts such as *ancestor*, *prime number*, *member of a list* and so on.

The acquisition of recursive and even mutually recursive concepts (such as *odd/even*) is explicitly addressed in the inductive logic programming system Atre (Malerba, 2003) and also covered by Igor2.

Ancestor (9 examples, 10.1 sec)
(and corresponding 4 examples for IsIn and Or)

```
Ancestor(X, Y, nil) = nilp .
Ancestor(X, Y, node(Z, L, R)) =
  IsIn(Y, node(Z, L, R))  if X == Z .
Ancestor(X, Y, node(Z, L, R)) =
  Ancestor(X, Y, L) Or Ancestor(X, Y, R)  if X =/= Z .
```

Corresponding to:

ancestor(x,y) = parent(x,y).

ancestor(x,y) = parent(x,z), ancestor(z,y).

isa(x,y) = directlink(x,y).

isa(x,y) = directlink(x,z), isa(z,y).

Fig. 7. Learned Transitivity Rules

In the following, we will focus on the concept of *ancestor* which is often used as standard example in inductive logic programming (Lavrač & Džeroski, 1994). The competence underlying the correct application of the *ancestor* concept, that is, correctly classifying a person as ancestor of some other person, in our opinion is the correct application of the transitivity relation in some partial ordering. We believe that if a person has grasped the concept of transitivity in one domain, such as *ancestor*, this person will also be able to correctly apply it in other, previously unknown domains. For example, such a person should be able to correctly infer *is-a* relations in some ontology. We plan to conduct a psychological experiment with children to strengthen this claim.

For simplicity of modeling, we used binary trees as domain model. For trees with arbitrary branching factor, the number of examples would have to be increased significantly. The transitivity rule learned by Igor2 is given in Figure 7.

Finally, we demonstrate the ability of Igor2 to learn a phrase-structure grammar. This problem is also addressed in grammar inference research (Sakakibara, 1997). We avoided the problem of learning word-category associations and provided examples abstracted from concrete words (see Figure 8). This, in our opinion, is legitimate since word categories are learned before complex grammatical structures are acquired. There is empirical evidence that children first learn rather simple Pivot grammars where the basic word categories are systematically positioned before they are able to produce more complex grammatical structures (Braine, 1963; Marcus, 2001). Learning grammar rules from example experience is also addressed in research on usage based theories of language acquisition (Tomasello, 2003).

The abstract sentence structures correspond to sentences as (Covington, 1994):

1: *The dog chased the cat.*
2: *The girl thought the dog chased the cat.*
3: *The butler said the girl thought the dog chased the cat.*

presented grammar
```
S -> NP VP
NP -> d n
VP -> v NP | v S
```
examples
```
fmod GENERATOR is
  *** types
  sorts Cat CList Depth .
  ops d n v : -> Cat [ctor] .
  op ! : -> CList [ctor] .
  op __ : Cat CList -> CList [ctor] .
  op 1 : -> Depth [ctor] .
  op s_ : Depth -> Depth [ctor] .
  *** target fun declaration
  op Sentence : Depth -> CList [metadata "induce"] .
  *** examples
  eq Sentence(1) = (d n v d n !) .
  eq Sentence(s 1) = (d n v d n v d n !) .
  eq Sentence(s s 1) = (d n v d n v d n v d n !) .
```
learned grammar rules (3 examples, 0.072 sec)
```
Sentence(1) = (d n v d n !)
Sentence(s N) = (d n v Sentence(N))
```

Fig. 8. Learning a Phrase-Structure Grammar

The recursive rules can generate sentences for an arbitrary depth which is given as parameter. Such center-embedded clauses are relational complex and parsing such sentences has similar demands than solving Tower of Hanoi puzzles (Halford, Wilson, & Phillips, 1998). Generation as well as reception of such sentences has high demand on working memory, because partial structures have to be held on a stack. However, again, we want to emphasize that we propose Igor2as an approach to learn rules representing the *competence* of a speaker or problem solver and not his or her performance.

Igor2 can also learn more complex rules, for example allowing for conjunctions of noun phrases or verb phrases. In this case, a nested numerical parameter can be used to specify at which position conjunctions in which depth can be introduced. Alternatively, a parser could be learned. Note that the learned rules are simpler than the original grammar but fulfill the same functionality.

6 Conclusion and Further Work

We presented the inductive programming system Igor as a mechanism for inducing generalized rules from example experience. Our focus was to demonstrate Igor's capability to induce generalized strategies for various domains in the area of problem solving. We could show that Igor learns rule sets for generating correct and complete action sequences from a *small amount* of *only positive* examples in *very short time*. This corresponds with claims in cognitive science about the human ability of generalization learning (Marcus, 2001; Klahr & Wallace, 1976).

Furthermore, we could demonstrate that Igor can not only be applied to rule learning in problem solving but also to learn generalized rules for reasoning and natural language processing. Since the induction mechanism underlying Igor is driven by analyzing structural regularities and thereby domain independent, Igor can be considered as one proposal for a general *cognitive rule acquisition device* realizing the claims Chomsky made about a language acquisition device (LAD) which allows to obtain a grammar for a language from linguistic experience (Chomsky, 1959, 1965). Unfortunately, this idea became quite unpopular (Levelt, 1976): One reason is, that only performance and not competence is empirically testable and therefore the idea was only of limited interest to psycho-linguists. Second, Chomsky (1959) argued that there "is little point in speculating about the process of acquisition without much better understanding of what is acquired" and therefore linguistic research focussed on search for a universal grammar. Third, the LAD is concerned with *learning* and learning research was predominantly associated with Skinner's reinforcement learning approach which clearly is unsuitable as a language acquisition device since it explains language acquisition as selective reinforcement of imitation.

Since the time of the original proposal of the LAD there was considerable progress in the domain of machine learning (Mitchell, 1997) and we propose that it might be worthwhile to give this plausible assumption of Chomsky a new chance. Obviously, typical approaches to classification learning from rather large sets of positive *and* negative examples which are based on the assumption of PAC (probably approximately correct) learnability are unsuitable to model a general rule acquisition device. To model the acquisition of a general cognitive competence, an approach is needed which learns from positive examples only, covers all examples correctly and learns a set of rules which capture the regularities observed in these examples. A niche of machine learning research where algorithmic approaches fulfilling these presuppositions are developed and investigated is inductive programming. Inductive programming is based on the notion of language identification in the limit (Gold, 1967) and can be viewed as more general approach to rule learning than grammar inference (Sakakibara, 1997) since it is concerned with learning recursive programs from examples. As argued above, especially analytical approaches to inductive programming might be helpful instruments to provide an algorithmic foundation for a general rule acquisition device.

Furthermore, the conception of inductive biases (Mitchell, 1997) introduced in machine learning, namely restriction (i.e. language) and preference (i.e. search) bias might be an alternative approach to the search of a universal grammar: Instead of providing a general grammatical framework from which each specific grammar – be it for a natural language or for some other problem domain – can be derived, it might be more fruitful to provide a set of constraints (biases) which characterize what kinds of rule systems are learnable by humans.

There are several aspects of IGOR which we want to explore in future research. One line of work will be concerned with different possibilities to include background knowledge. Currently, background knowledge can – but needs not – be presented to IGOR in form of additional examples. We used this possibility, for example, when generating the rule set for *tower* by presupposing knowledge about how to check whether a block is *clear* and how to check whether blocks in a partial tower are already in correct sequence *isTower*. Assuming that humans exploit already acquired knowledge to solve more complex domains, we want to provide a possibility to present background knowledge which has already the form of recursive rule sets. Furthermore, we currently work on an extension of IGOR2 where examples are checked for their subsumption under higher-order program schemes (Hofmann & Kitzelmann, 2010). We could already demonstrate that by this technique, problems which previously could only dealt with if background knowledge was given to IGOR2 now can be learned without recurring to background knowledge. To investigate mechanisms for constructing examples suitable for IGOR we currently are working on embedding IGOR in a robot scenario where the robot learns recursive rule sets for behavior which will generate positive reinforcements. Here we are experimenting with color patterns with underlying grammar structures such as $(red)^n (green)(red)^n$. Only if the robot accesses patterns belonging to this grammar it will be rewarded. From some examples, it should generalize the grammar rules which allow it to select rewarding patterns of arbitrary complexity.

References

Altmann, E. M., & Trafton, J. G. (2002). Memory for goals: An activation-based model. *Cognitive Science*, *26*, 39-83.

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

Anderson, J. R., Albert, M. V., & Fincham, J. (2005). Tracing problem solving in real time: fMRI analysis of the subject-paced tower of hanoi. *Journal of Cognitive Neuroscience*, *17*, 1261-1274.

Anderson, J. R., & Douglass, S. (2001). Tower of Hanoi: Evidence of the cost of goal retrieval. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *27*(6), 1331-1346.

Anderson, J. R., & Lebière, C. (1998). *The atomic components of thought.* Mahwah, NJ: Lawrence Erlbaum.

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

Baars, B. J., & Franklin, S. (2003). How conscious experience and working memory interact. *TRENDS in Cognitive Sciences*, *7*(4), 166-172.

Bach, J. (2009). *Principles of synthetic intelligence. Psi, an architecture of* *motivated cognition.* New York: Oxford University Press.

Biermann, A. W., Guiho, G., & Kodratoff, Y. (Eds.). (1984). *Automatic program construction techniques.* New York: Macmillan.

Braine, M. (1963). On learning the gramamtical order of words. *Psychological Review*, *70*, 332-348.

Bruner, J. S., Goodnow, J. J., & Austin, G. A. (1956). *A study of thinking.* New York: Wiley.

Bühler, K. (1907). Tatsachen und Probleme zu einer Psychologie der Denkvorgänge. Über Gedanken. *Archiv für Psychologie*, *9*, 297-365.

Chomsky, N. (1959). Review of Skinner's 'Verbal Behavior'. *Language*, *35*, 26-58.

Chomsky, N. (1965). *Aspects of the theory of syntax.* Cambridge, MA: MIT Press.

Covington, M. A. (1994). *Natural language processing for Prolog programmers.* Prentice Hall.

Flener, P., & Schmid, U. (2009). An introduction to inductive programming. *Artificial Intelligence Review*, *29*(1), 45-62.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated planning: Theory and practice.* Morgan Kaufmann.

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

Goodman, N. (1965). *Fact, fiction, and forecast.* Indianapolis: Bobbs-Merrill.

Green, C. (1969). Application of theorem proving to problem solving. In *Proc. 1st International Joint Conference on Artificial Intelligence (IJCAI-69)* (p. 342-344).

Halford, G. S., Wilson, W. H., & Phillips, S. (1998). Processing capacity defined by relational complexity: Implications for comparative, developmental, and cognitive psychology. *Behavioral and Brain Sciences*, *21*, 803-865.

Hammer, B., & Hitzler, P. (Eds.). (2007). *Perspectives of neural-symbolic integration.* Berlin: Springer.

Hofmann, M., & Kitzelmann, E. (2010). I/O guided detection of list catamorphisms – towards problem specific use of program templates in ip. In J. Gallagher & J. Voigtländer (Eds.), *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (p. 93-100). New York: ACM SIGPLAN.

Hofmann, M., Kitzelmann, E., & Schmid, U. (2008). Analysis and evaluation of inductive programming systems in a higher-order framework. In A. Dengel, K. Berns, T. M. Breuel, F. Bomarius, & T. R. Roth-Berghofer (Eds.), *KI 2008: Advances in Artificial Intelligence* (p. 78-86). Berlin: Springer.

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

Hunt, E., Marin, J., & Stone, P. J. (1966). *Experiments in induction.* New

York: Academic Press.

Kahney, H. (1989). What do novice programmers know about recursion? In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (p. 209-228). Lawrence Erlbaum.

Kaplan, C., & Simon, H. (1990). In search of insight. *Cognitive Psychology*, *22*, 374-419.

Katayama, S. (2005). Systematic search for lambda expressions. In *Trends in functional programming* (p. 111-126).

Kitzelmann, E. (2003). *Inductive Functional Programming - a Term-Construction and Folding Approach.* Unpublished master's thesis, Dept. of Computer Science, TU Berlin.

Kitzelmann, E. (2009). Analytical inductive functional programming. In M. Hanus (Ed.), *Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008, Valencia, Spain)* (Vol. 5438, p. 87-102). Springer.

Kitzelmann, E. (2010). *A combined analytical and search-based approach to the inductive synthesis of functional programs.* Unpublished doctoral dissertation, Fakultät Wirtschaftsinformatik und Angewandte Informatik, Otto-Friedrich Universität Bamberg.

Kitzelmann, E., & Schmid, U. (2006). Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, *7*(Feb), 429–454.

Klahr, D., & Wallace, J. (1976). *Cognitive development: An information processing view.* Hillsdale, NJ: Erlbaum.

Knoblich, G., Ohlsson, S., Haider, H., & Rhenius, D. (1999). Constraint relaxation and chunk decomposition in insight. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *25*(6), 1534-1556.

Korf, R. E. (1985). Macro-operators: a weak method for learning. *Artificial Intelligence, 1985*, *26*, 35–77.

Kotovsky, K., Hayes, J. R., & Simon, H. A. (1985). Why are some problems hard? Evidence from Tower of Hanoi. *Cognitive Psychology*, *17*(2), 248-294.

Koza, J. (1992). *Genetic programming: On the programming of computers by means of natural selection.* Cambridge, MA: MIT Press.

Lavrač, N., & Džeroski, S. (1994). *Inductive logic programming: Techniques and applications.* London: Ellis Horwood.

Levelt, W. (1976). *What became of LAD?* Lisse: Peter de Ridder Press.

Malerba, D. (2003). Learning recursive theories in the normal ILP setting. *Fundamenta Informaticae*, *57*(1), 39-77.

Manna, Z., & Waldinger, R. (1987). How to clear a block: a theory of plans. *Journal of Automated Reasoning*, *3*(4), 343–378.

Marcus, G. F. (2001). *The algebraic mind. integrating conncetionism and cognitive science.* Cambridge, MA: Bradford.

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)* (pp. 667–677). San Francisco, CA: Morgan Kaufmann.

McCarthy, J. (1963). *Situations, actions, and causal laws* (Memo No. 2). Stanford, California: Stanford University Artificial Intelligence Project.

Meyer, R. (1992). *Thinking, problem solving, cognition, second edition.* Freeman.

Minton, S. (1985). Selectively generalizing plans for problem-solving. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI-85)* (pp. 596–599). San Francisco, CA: Morgan Kaufmann.

Mitchell, T. M. (1997). *Machine learning.* New York: McGraw-Hill.

Newell, A. (1982). The knowledge level. *Artificial Intelligence*, *18*, 87-127.

Ohlsson, S. (1984). Restructuring revisited: Summary and critique of the Gestalt theory of problem solving. *Scandinavian Journal of Psychology*, *25*, 65-78.

Olsson, R. (1995). Inductive functional programming using incremental program transformation. *Artificial Intelligence*, *74*(1), 55-83.

Plotkin, G. D. (1969). A note on inductive generalization. In *Machine intelligence* (Vol. 5, pp. 153–163). Edinburgh University Press.

Quinlan, J., & Cameron-Jones, R. (1995). Induction of logic programs: FOIL and related systems. *New Generation Computing, Special Issue on Inductive Logic Programming*, *13*(3-4), 287–312.

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.

Sakakibara, Y. (1997). Recent advances of grammatical inference. *Theoretical Computer Science*, *185*, 15-45.

Schmid, U. (2003). *Inductive synthesis of functional programs – Learning domain-specific control rules and abstract schemes.* Heidelberg: Springer.

Schmid, U., Kitzelmann, E., & Plameijer, R. (Eds.). (2010). *Approaches and Applications of Inductive Programming Third International Workshop, AAIP 2009, Edinburgh, UK, September 4, 2009, Revised Papers.* Heidelberg: 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)* (pp. 371–378). Menlo Park, CA: AAAI Press.

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.

Simon, H. (1958). Rational choice and the structure of the environment. In *Models of bounded rationality* (Vol. 2). Cambridge, MA: MIT Press.

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

Sun, R. (2007). The importance of cognitive architectures: An analysis based on CLARION. *Journal of Experimental and Theoretical Artificial Intelligence*, *19*(2), 159-193.

Tenenbaum, J., Griffiths, T., & Kemp, C. (2006). Theory-based Bayesian models of inductive learning and reasoning. *Trends in Cognitive Sciences*, *10*(7), 309-318.

Tomasello, M. (2003). *Constructing a language: A usage-based theory of language acquisition.* Harvard University Press.

Veloso, M. M., & Carbonell, J. G. (1993). Derivational analogy in Prodigy: Automating case acquisition, storage, and utilization. *Machine Learning*, *10*, 249–278.

Welsh, M. C., & Huizinga, M. (2005). Tower of Hanoi disk-transfer task: Influences of strategy knowledge and learning on performance. *Learning and Individual Differences*, *15*, 283–298.