Analytical Inductive Programming as a Cognitive Rule Acquisition Devise

Ute Schmid and Martin Hofmann and Emanuel Kitzelmann
Faculty Information Systems and Applied Computer Science
University of Bamberg, Germany
{ute.schmid, martin.hofmann, emanuel.kitzelmann}@uni-bamberg.de

Abstract
One of the most admirable characteristic of the human cognitive system is its ability to extract generalized rules covering regularities from example experience presented by or experienced from the environment. Humans' problem solving, reasoning and verbal behavior often shows a high degree of systematicity and productivity which can best be characterized by a competence level reflected by a set of recursive rules. While we assume that such rules are different for different domains, we believe that there exists a general mechanism to extract such rules from only positive examples from the environment. Our system **Igor2** is an analytical approach to inductive programming which induces recursive rules by generalizing over regularities in a small set of positive input/output examples. We applied **Igor2** to typical examples from cognitive domains and can show that the **Igor2** mechanism is able to learn the rules which can best describe systematic and productive behavior in such domains.

Introduction
Research in inductive programming is concerned with the design of algorithms for synthesis of recursive programs from incomplete specifications such as input/output examples of the desired program behavior, possibly together with a set of constraints about size or time complexity (Biermann, Guiho, & Kodratoff 1984; Flener 1995). In general, there are two distinct approaches to inductive programming – search-based generate-and-test algorithms (Olsson 1995; Quinlan & Cameron-Jones 1995) and data-driven analytical algorithms (Summers 1977; Kitzelmann & Schmid 2006). In the first case, given some language restriction, hypothetical programs are generated, tested against the specification and modified until they meet some given criteria. In the second case, regularities in the input/output examples are identified and a generalized structure is built over the examples. While search-based approaches – in principle – can generate each possible program and therefore might be able to find the desired one given enough time, analytical approaches have a more restricted language bias. The advantage of analytical inductive programming is that programs are synthesized very fast, that the programs are guaranteed to be correct for all input/output examples and fulfill further characteristics such as guaranteed termination and being minimal generalizations over the examples. The main goal of inductive programming research is to provide assistance systems for programmers or to support end-user programming (Flener & Partridge 2001).

From a broader perspective, analytical inductive programming provides algorithms for extracting generalized sets of recursive rules from small sets of positive examples of some behavior. Such algorithms therefore cannot only be applied to input/output examples describing the behavior of some program but to arbitrary expressions. Taking this standpoint, analytical inductive programming provides a general device for the acquisition of generalized rules.

In the sixties, Chomsky proposed that the human mind possesses a language acquisition device (LAD) which allows us to extract grammar rules from the example language experience we are exposed to (Chomsky 1959; 1965). Input to this device are the linguistic experiences of a child, output is a grammar reflecting the linguistic competence of the individuum. Unfortunately, this postulation became quite unpopular for several reasons (Levelt 1976): One reason is, that only performance and not competence is empirically testable and therefore the idea was only of limited interest to psycholinguists. Second, Chomsky argued that there “is little point in speculating about the process of acquisition without much better understanding of what is acquired” (Chomsky 1959) 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

---

*Research was supported by the German Research Community (DFG), grant SCHM 1239/6-1.
Copyright © 2008, The Second Conference on Artificial General Intelligence (AGI-09.org). All rights reserved.*
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.

In the following we give a short overview of our inductive programming system Igor2 together with its biases. Then we illustrate Igor2’s ability as a cognitive rule acquisition device in the domains of problem solving, reasoning, and natural language processing.  

Recursive Structure Generalization

Igor2 (Kitzelmann 2008) was developed as a successor to the classical Thesys system for learning Lisp programs from input/output examples (Summers 1977) and its generalization Igor1 (Kitzelmann & Schmid 2006). To our knowledge, Igor2 is currently the most powerful system for analytical inductive programming. Its scope of inducable programs and the time efficiency of the induction algorithm compares well 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.

Igor2 specifications consist of: a small set of positive input/output examples, presented as equations, which have to be the first examples with respect to the underlying data type and a specification of the input data type. Furthermore, background knowledge for additional functions can (but must 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 of the example equations. If a rule contains unbound variables, 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 righthand side of the rule; (ii) replacing the righthand side of a rule by a (recursive) call to a defined function (including the target 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 subterms in the righthand side of a rule which contain unbound variables by a call to new subprograms.

Problem Solving

Often, in cognitive psychology, speed-up effects in problem solving are modelled simply 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). 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). 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 on the problems Clearblock and Rocket faster and without specific assumptions to preprocessing and furthermore can tackle more complex problem domains such as building a tower in the blocksworld domain.

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. We illustrate the idea of our approach with the simple clearblock problem (see figure 1). 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 puttable(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 notation for the problem domain and the problem descriptions. 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 – the puttable operator – and problem descriptions given in figure 1.

The resulting action sequences can be obtained by any PDDL planner (Ghallab, 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 (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 2). 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 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.

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).

A most challenging problem domain which is still used as a benchmark for planning algorithms is blocksworld. A typical blocksworld 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 then 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 2 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 3. The goal is to construct a tower for some predefined ordering of blocks. To represent this ordering, blocks are represented constructively as “successors” to the table with respect to the goal state. Therefore the top object of the to be constructed tower 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 $s$ $s$ $s$ table. The second argument again is a situation variable which initially holds the initial state. In the example in figure 3 $s s$ $s$ 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 outmost 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.

Finally, the recursive solution to the Tower of Hanoi problem was generated by Igor2 from three examples (see figure 2). The input to Igor2 is given in figure 4.

For the discussed typical problem solving domains Igor2 could infer the recursive 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.

### Reasoning

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
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))))))) .

Figure 4: Posing the Tower of Hanoi problem for 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).

Figure 5: Learned Transitivity Rules

apply recursive concepts such as ancestor, prime number, member of a list and so on.

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 5.

Natural Language Processing

Finally, we come back to Chomsky’s claim of an LAD. We presented Igor2 with examples to learn a phrase-structure grammar (in the very original grammar, d n v are non-terminals D N V which go to concrete words)

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 s_ : -> Depth [ctor] .

*** target fun declaration
op Sentence : Depth -> CList [metadata "induce"] .

*** examples
eq Sentence(1) = (d n v d n !) .
eq Sentence(s N) = (d n v Sentence(N)) .
eq Sentence(s s 1) = (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))

Figure 6: Learning a Phrase-Structure Grammar

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 6). 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).

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.
4: The gardener claimed the butler said the girl thought the dog chased the cat.

The recursive rules can generate sentences for an arbitrary depth which is given as parameter. 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.

Conclusion

Igor2 is a rather successful system for analytical induc-
tive programming. Up to now we applied Igor2 to
typical programming problems (Hofmann, Kitzelmann,
& Schmid 2008). In this paper we showed that analytical
inductive programming is one possible approach to
model a general cognitive rule acquisition device and
we successfully applied Igor2 to a range of prototypical
problems from the domains of problem solving, reason-
ning, and natural language processing. Analytical in-
ductive programming seems a highly suitable approach
to model the human ability to extract generalized rules
from example experience since it allows fast general-
ization from very small sets of only positive examples
(Marcus 2001). In our opinion, it would be worth-
while including such a mechanism into general cognitive
architectures to model rule acquisition on a symbolic
level. From the perspective of universal grammar re-
search, it would be interesting to obtain empirical data
which can give hints about which types of rule systems
can be learned from what example sets in what time
and thereby get new insights in the language bias of
human learners.

References
Anderson, J. R., and Lebière, C. 1998. The atomic com-
Anzai, Y., and Simon, H. 1979. The theory of learning by
Biermann, A. W.; Guilo, G.; and Kodratoff, Y., eds. 1984.
Automatic Program Construction Techniques. New York:
Macmillan.
Braine, M. 1963. On learning the grammatical order of
Brainer, J. S.; Goodnow, J. J.; and Austin, G. A. 1956. A
Study of Thinking. New York: Wiley.
Language 35:26–58.
Chomsky, N. 1965. Aspects of the Theory of Syntax. Cam-
bidge, MA: MIT Press.
Covington, M. A. 1994. Natural Language Processing for
Prolog Programmers. Prentice Hall.
Flener, P., and Partridge, D. 2001. Inductive program-
Flener, P. 1995. Logic Program Synthesis from Incomplete
Planning: Theory and Practice. Morgan Kaufmann.
Gold, E. 1967. Language identification in the limit. Infor-
mation and Control 10:447–474.
Hofmann, M.; Kitzelmann, E.; and Schmid, U. 2008. An-
alysis and evaluation of inductive programming systems in
a higher-order framework. In Dengel, A. et al., eds., KI
2008: Advances in Artificial Intelligence, number 5243 in
LNAI, 78–86. Berlin: Springer.
Hunt, E.; Marin, J.; and Stone, P. J. 1966. Experiments in
Kitzelmann, E., and Schmid, U. 2006. Inductive synthe-
sis of functional programs: An explanation based general-
ization approach. Journal of Machine Learning Research
7(Feb):429–454.
Kittelmann, E. 2008. Analytical inductive functional pro-
gramming. In Hans, M., ed., Pre-Proceedings of LOPSTR
Korf, R. E. 1985. Macro-operators: a weak method for learn-
of computers by means of natural selection. Cambridge,
MA: MIT Press.
Lavrač, N., and Džeroski, S. 1994. Inductive Logic Pro-
gramming: Techniques and Applications. London: Ellis
Horwood.
Press.
Manna, Z., and Waldinger, R. 1987. How to clear a block: a
378.
Marcus, G. F. 2001. The Algebraic Mind. Integrating Con-
policies in planning using concept languages. In Proc. KR
McCarthy, J. 1963. Situations, actions, and causal
laws. Memo 2, Stanford University Artificial Intelligence
Project, Stanford, California.
Minton, S. 1985. Selectively generalizing plans for problem-
Morgan Kaufmann.
McGraw-Hill.
Olsson, R. 1995. Inductive functional programming using
incremental program transformation. Artificial Intelligence
Quinlan, J., and Cameron-Jones, R. 1995. Induction of
logic programs: FOIL and related systems. New Genera-
Rosenbloom, P. S., and Newell, A. 1986. The chunking
of goal hierarchies: A generalized model of practice. In
Michalski, R. S.; Carbonell, J. G.; and Mitchell, T. M.,
eds., Machine Learning - An Artificial Intelligence
Sakakibara, Y. 1997. Recent advances of grammatical
programm synthesis to macro learning. In Proc. AIPS
Shavlik, J. W. 1990. Acquiring recursive and iterative con-
cepts with explanation-based learning. Machine Learning
5:39–70.
framework for composing disjunctive and iterative macro-
Summers, P. D. 1977. A methodology for LISP program
analogy in Prodigy: Automating case acquisition, storage,