Induction of Recursive Program Schemes

Ute Schmid & Fritz Wysotzki

Department of Computer Science
Technical University Berlin
email: schmid,wysotzki@cs.tu-berlin.de

Abstract. In this paper we present an approach to the induction of recursive structures from examples which is based on the notion of recursive program schemes. We separate induction from examples in two stages: (1) constructing initial programs from examples and (2) folding initial programs to recursive program schemes. By this separation, the induction of recursive program schemes can be reduced to a pattern-matching problem which can be handled by a generic algorithm. Construction of initial programs is performed with an approach to universal planning. "Background knowledge" is given in the form of operators and their conditions of application. Furthermore synthesizing recursive program schemes instead of programs in a predefined programming language enables us to combine program synthesis and analogical reasoning. A recursive program scheme represents the class of structural identical programs and can be assigned different semantics by interpretation. We believe that our approach mimicks in some way the problem solving and learning behavior of a (novice) human programmer and that our approach integrates theoretical ideas and empirical results of learning by doing and learning by analogy from cognitive science in a unique framework.

Keywords: Inductive Program Synthesis, Planning and Learning, Analogy, Cognitive Modelling

1 Introduction

Building recursive definitions from examples is an old topic in automatic program construction [3]. Such techniques can be exploited in different fields as automatic theorem proving or generation of (robot) action sequences. There was a lot of interest on inductive approaches to the synthesis of recursive programs during the seventies and eighties in the context of functional (LISP) programming which is now revived in inductive logic programming (ILP; [9]). The general idea of inductive program synthesis is to generalize over the structures of a set of (positive) examples with a kind of unsupervised learning algorithm.

We propose an approach to inductive program synthesis which differs from the LISP-based methods and from ILP in two aspects: Firstly, we take a more abstract view on programs, that is, we describe programs as elements of a term algebra and we infer recursive program schemes (RPSs; cf. [6,7]) instead of LISP functions or PROLOG clauses. Secondly, we separate the problem of program synthesis from examples in two distinct processes: we deal with the construction of "Summers"-like initial programs from examples [18, 20] with methods of "universal planning" [17, 21] and use the initial programs as input to our program synthesis algorithm.

We believe that our approach mimicks certain aspects of human (novice) programmers (cf. [2]). Building initial programs from examples by planning can be viewed as constructing the minimal sequence of operations by which the example input can be transformed into the desired output (i.e. goal state) by using only primitive functions (i.e. predefined operators). Combining different simulation traces in one planning tree can be viewed as integrating experience with different inital states of a problem (i.e. examples) into a single structure which is composed of conditional expressions. This process corresponds roughly to compilation/chunking of rules in cognitive models of skill acquisition (cf. [1]). Using the planning tree (initial program) as input to program synthesis describes a second stage of learning, generalization over recursive enumerable problem spaces. This stage of skill acquisition sketches the learning of a specific problem solving strategy. Using RPSs rather than a programming language enables us to model a third stage of learning, generalization over classes of programs, by abstracting from the concrete semantics of the symbols contained in an RPS.

Let's illustrate this idea with an example: A novice programmer or our system is confronted with the problem to write a program to sum the elements of a list of natural numbers. In a first step he/it constructs sequential solution sequences for example lists of length zero, one, two and three:

1. *if empty(l) then 0,*
2. *if empty (tail(l)) then head(l),*
3. *if empty(tail(tail(l))) then plus(head(l), head(tail(l))),*
4. *if empty(tail(tail(tail(l)))) than plus(head(l), plus(head(tail(l)), head(tail(tail(l))))).*

These traces (describing cases mutually excluding each other) are integrated in a kind of universal plan representing the initial program:

```
if empty(l) then 0
  else if empty(tail(l)) then head(l)
    else if empty(tail(tail(l))) then plus(head(l), head(tail(l)))
      else if empty(tail(tail(tail(l))))
        then plus (head(l), plus(head(tail(l)), head(tail(tail(l))))).
```

The experience with lists up to length three provides the basis for generalization to lists of arbitrary length n, that is for inferring an RPS: *sumlist(l) = if empty(l) then 0 else plus(head(l), sumlist(tail(l)))*. The infered RPS is stored in memory and can be used to solve structural identical problems with different semantics by analogical reasoning, for example *sum(x) = if eq0(x) then 0 else plus(succ(x), sum(pred(x)))*.

In this paper - in contrast to other work on inductive program synthesis - we do not focus on the usual list or number problems (as the programs given above) but we will concentrate on blockworld problems (cf. [13]). Thereby we hope to bridge the gap between planning and inductive programming. Especially, our approach could help to overcome the complexity problems in universal planning [10] by restricting planning to problems of small complexity and using inductive program synthesis to scale up. In section 4.2 will be shown, that using blockworld problems (and interpreting them as RPSs) does not restrict the class of recursive programs which can be inferred to this domain. For most list or number problems there are structural equivalent blockworld problems. A blockworld problem corresponding to the linear structure of the examples given above is for example to clear a given block x in a tower: *clearblock(x,s) = if cleartop(x) then s else puttable(topof(x), clearblock(topof(x),s))*, where s is a situation variable.

In the following we will first describe work which is related to our approach. Afterwards we will introduce the concept of recursive program schemes and describe our induction algorithm together with examples of synthetizised structures. Than we will shortly illustrate our use of planning and analogical reasoning in inductive program syntheses. The paper finishes with a conclusion and plans for further work.

2 Related Work

Our approach has its background in the work on synthesis of functional programs (esp. [11, 18]). That the methodologies introduced in the seventies are apt to handle the benchmarks for recursive program synthesis given by ILP was shown on the example of the BMW algorithm [4, 11].

The original idea of inferring RPSs rather than LISP functions and of separating the generation of initial programs from program synthesis was presented in [20] and [21]. In contrast to most work in ILP (cf. [9]), we use only a small number of positive examples (also of interest now in ILP; cf. [14]). By restricting the synthesis task to generalizing over initial programs we have no need of regarding background knowledge, that is, our work is similar to the generalization-to-n approaches in grammatical inference (cf. [5]). That is, by splitting program synthesis in construction of initial programs (where domain specific background knowledge is used) and folding initial programs to RPSs we can reduce the synthesis algorithm to pattern-matching which can be performed with a generic algorithm.

Our main interest is not to present theoretical results on inductive synthesis of recursive structures but to make these ideas useful in the context of planning and problem solving. While [13] presented a formal approach to deductive reasoning in planning, we want to introduce inductive program synthesis as an alternative approach to learning in the domain of plan construction as for example employed in PRODIGY (cf. [19]). Finally, we believe, that our approach integrates ideas and empirical findings in the area of skill acquistion as introduced in cognitive science (cf. [1]). The notion of RPSs enables us to model the acquisition of problem solving schemes and their use in analogical reasoning in a unified way (see [16]).

3 Recursive Program Schemes (RPSs)

The theoretical framework for induction of RPSs was presented in [6], which we use as background for our approach. In the following we give some basic definitions before introducing the concept of an RPS.

Definition 1 (Term Algebra) *Let V be a set of variables and F a set of function symbols with $F^i \subseteq F$ as set of function symbols with arity i and $\Omega \in F$ as symbol for "undefined". Then $M(V, F)$ is the set of all welldefined terms:*
1. *$v \in V$ and $c \in F^0$ (constant symbols) are terms.*
2. *if $t_1 \ldots t_n$ are terms and $f \in F^n$ is a function symbol of arity n then $f(t_1 \ldots t_n)$ is a term.*

Definition 2 (Extended Term Algebra) *Let $\Phi = \{G_1, \ldots G_n\}$ be a set of function variables with arity $G_i = k_i$. Then we call $M(V, F \cup \Phi)$ the extended term algebra.*

Definition 3 (Partial Order over M) *The term algebra can be extended to include infinite terms by defining a partial order over it. Ω is the bottom element and we define $t < t'$ if t' can be generated from t by replacing Ω in t by a term $\neq \Omega$ (but which may contain the symbol Ω). Every ordered subset $A \subseteq M$ has an upper limit $Sup(A)$ which can be infinite.*

Definition 4 (McCarthy-Conditional) *The McCarthy-Conditional g(x,y,z) is a function of arity three which can be interpreted as conditional expression (if x then y else z).*

Definition 5 (Recursive Program Scheme) *An RPS is a pair $<\Sigma, t>$ with $\Sigma =< G_i(v_1 \ldots v_n) = t_i \mid i = 1 \ldots n >$ is a system of equations ("subroutines") and $t \in M$ is the "main program".*

If G_i is contained in t_i, the equation defines a recursive function.

To illustrate the notion of an RPS we give an example. We define the extended term algebra as $M(\{x, s\}, \{\Omega, cleartop^1, topof^1, puttable^2\} \cup \{clearblock^2\})$. We define Σ as *clearblock(x,s) = g(cleartop(x), s, puttable(topof(x), clearblock(topof(x), s)))* and $t = clearblock(x, s)$. That is, our Σ consists of a single equation. Note that the lefthand side of the equation corresponds to the head of a function, the righthand side to the body. We have terms in M with $\Omega <$ *g(cleartop(x), s, puttable(topof(x), Ω))* $<$ *g(cleartop(x), s, puttable(topof(x), g(clearblock(topof(x), s, puttable(topof(topof(x)), Ω)))))* $< \ldots$.

In a similar way as in eval-apply interpreters, a system of equations (functions) can be solved, by replacing the name of a user-defined function (i.e. a function variable) by its body:

Definition 6 (Rewrite Rule) *An expression can be transformed by the rewrite rule Θ in the following way:*
1. *$\Theta(x) = x$ if $x \in V \cup F^0$*
2. *$\Theta(f(t_1 \ldots t_k)) = f(\Theta(t_1) \ldots \Theta(t_k))$*
3. *$\Theta(G(t_1 \ldots t_k)) = \theta(G)_{[\Theta(t_1)/v_1 \ldots \Theta(t_k)/v_k]}$.*
   That is, the head of G is replaced by its body $\theta(G)$ where all variables v_i are substituted by the rewritten terms t_i with which G was "called". Note, that with an expression $[t/v]$ we denote the substitution of a variable v by a term t (often written as $v \leftarrow t$).

We can use the rewrite rule given above to expand an RPS to a certain length l. A sequence of expansions from $l = 0, 1, 2 \ldots$ is called Kleene-sequence:

Definition 7 (Kleene-Sequence) *Let $G_i = t_i \in \Sigma$ be a recursive equation. We can define a sequence $\mathcal{G}_i^l$ for $l = 0, 1, 2 \ldots$ by: $\mathcal{G}_i^0 = \Omega$, $\mathcal{G}_i^l = \Theta(t_i) = t_i[\mathcal{G}_1^{l-1}/G_1 \ldots \mathcal{G}_n^{l-1}/G_n]$. That is, an expansion of length l can be constructed by substituting function calls G_i in t by expansions $\mathcal{G}_i^{l-1}$.*

For the example given above we can expand $clearblock(x, s)$ by

$\mathcal{G}^0 = \Omega$
$\mathcal{G}^1 = g(cleartop(x), s, puttable(topof(x), \Omega))$
$\mathcal{G}^2 = g(cleartop(x), s, puttable(topof(x),$
$\quad g(cleartop(topof(x)), s, puttable(topof(topof(x)), \Omega))))$
$\ldots$
$\mathcal{G}^l = g(cleartop(x), s, puttable(topof(x), \mathcal{G}^{l-1}_{[topof(x)/x]}))$.

For the Kleene-sequence it can be shown that $\mathcal{G}^{l-1} < \mathcal{G}^l$ and that $Sup(\mathcal{G}^l)$ is the least fixpoint of Σ (see fixpoint semantics; cf. [8, App. B]).

Up to now we only have regarded the syntactical aspects of RPSs. To calculate a value for an RPS (i.e. to compute a result), the symbols of the RPS have to be interpreted by functions and values have to be assigned to the variables:

Definition 8 (Interpretation and Valuation)
- *Interpretation I of an RPS: Function and constant symbols in the RPS are interpreted by functions and constants of a domain model with corresponding arity (and possibly types).*
- *Valuation β of variables: Each variable occuring in the (head of the) RPS has to be assigned a value (corresponding to variable type).*
- *In the following we will regard untyped structures only.*

4 Synthesis of RPSs from Initial Progams

4.1 An Algorithm for Induction of RPSs

Our approach to inductive synthesis of RPSs is based on the idea of the Kleene-sequence given above. That is, we reverse the process of expanding an RPS as proposed originally by [20]. We regard a given initial program as element of some Kleene-sequence which we try to identify and than fold the initial program to an RPS. Note, that our approach currently is restricted to infer a *single* recursive equation.

Definition 9 (Induction of a (linear) RPS) *Let $G \in M$ be an initial program. Then G can be folded into an RPS if G can be segmented in a sequence $\mathcal{G}^0 = \Omega$, $\mathcal{G}^l = tr(\mathcal{G}^{l-1}_{[\mathbf{t}/\mathbf{v}]}/m)$, where $\mathbf{t}$ is a vector of terms by which the vector $\mathbf{v}$ of variables in tr is replaced and where m is the place in term tr where the substitution by $\mathcal{G}^{l-1}$ is performed.*
If the complete initial program G can be described equations $\mathcal{G}^l$, that is if $G = \mathcal{G}^{l^*}$ for some l^*, we assume, that we can generalize $\mathcal{G}^{l^*}$ to an infinite sequence. That is, we extrapolate the RPS $\mathcal{G} = tr(\mathcal{G}_{[\mathbf{t}/\mathbf{v}]}/m)$.

Let's go back to our *clearblock*-example for illustrating this idea: Input into the synthesis algorithm may be the following initial program, defined for one up to three-block towers:

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

We can segment G into $\mathcal{G}^0$, $\mathcal{G}^1$, $\mathcal{G}^2$ as shown in section 3 and in $\mathcal{G}^3 = G$. G can be folded by induction into an RPS according to definition 9:

$$\mathcal{G} = g(cleartop(x), s, puttable(topof(b), \mathcal{G}_{[topof(x)/x]})).$$

This RPS generalizes the experience with clearing the bottom block of towers consisting of one up to three blocks to towers of arbitrary height.

The definition given above covers all structures with a single recursive call only, that is tail recursion and linear recursion (see figure 2). To deal with structures with more than one recursion point and with tree recursion (see section 4.2), the definition can be extended:

Definition 10 (Induction of an RPS) *Let $G \in M$ be a initial program. Then G can be folded into an RPS if G can be segmented in a sequence $\mathcal{G}^0 = \Omega$, $\mathcal{G}^l = tr(\mathcal{G}^{l-1}_{[t_1/v]}/m_1 \ldots \mathcal{G}^{l-1}_{[t_n/v]}/m_n)$ with m_i as positions in tr where the substitution by $\mathcal{G}^{l-1}$ is performed. If l is sufficiently high, we can extrapolate the RPS*

$$\mathcal{G} = tr(\mathcal{G}_{[t_1/v]}/m_1 \ldots \mathcal{G}_{[t_n/v]}/m_n).$$

An operational method for performing the induction of an RPS is given in algorithm 1. To find an RPS according to the definition above, we have to construct hypothetical segmentations of an initial program G and check the current hypothesis tr by matching it with subexpressions of G. For a given tr we than have to determine substitutions σ which hold for G. Our aim is to find the simplest hypothesis to fold the initial program G into an RPS. Therefore we enumerate the hypotheses in the order given in algorithm 1.

Algorithm 1 (Induction of an RPS)

– **Find structure *tr*:**

For all the following hypotheses tr of the recursive structure of G, the hypothesis holds if G can be segmented into subexpressions which are unifyable with tr

1. Assume tr starts at the root node, consists of a single conditional expression and of one recursion point only
2. If this assumption fails: enlarge the number of conditional expressions contained in tr (from two up to half of the number of occurences of the symbol g in G)
3. If these assumptions fail: assume a constant initial part in G and move the starting point for looking for tr from the root to another occurence of g (from the second position of g in G up to $n - 2$ if n is the number of occurences of g in G)

4. If these assumptions fail: assume that there exists more than one recursion point in G and start with assuming two recursion points and enlarge the hypothesis up to half of the number of occurences of g in G

– **Find substitutions σ_i:**

1. If the hypothesis consists of a single recursion point, generate a hypothesis for σ by unifying tr with that subexpression of G at which tr occurs for the second time
2. If the hypothesis consists of more than one recursion point, generate separate hypotheses σ_i for each recursion point by unifying tr with that subexpression t_i of G where tr ouccures the i-th time.

The expansion of tr is restricted by the number of conditional expressions of which an initial program G is composed. For example, we restrict the number of conditional expressions in tr to half the number of conditional expressions contained in G. Otherwise, we would construct a hypothesis which could not be validated for the given G. An illustration of algorithm 1 with the *clearblock*-problem is given in figure 1. The procedure can be made more intuitive if we represent G as a tree.

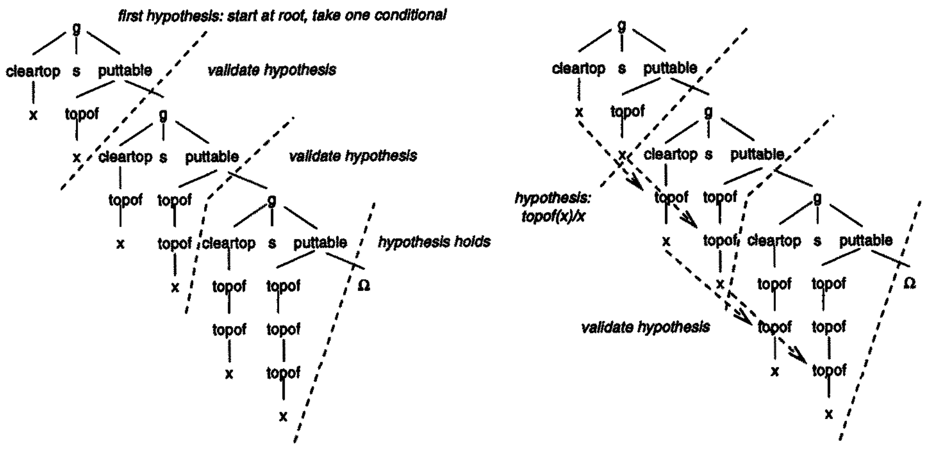

Fig. 1. Illustration of algorithm 1

Algorithm 1 was implemented and tested for a variety of recursive structures. The recursive structure tr can be composed of one ore more conditional expression and the possibility of a constant part in G not contained in the recursive structure is included in the algorithm. Up to now we are only dealing with cases, where the substitutions of variables are independent from each other. The separation of "find structure" and "find substitution" is introduced because we are planning to extend our substitution algorithm to cases of dependent variables. This extension is needed to deal with problems like the sorting of lists, where a counter j is substituted in relation to a counter i (i.e. we have a "nested loop").

4.2 Performance of the Algorithm

We will give some examples to illustrate the performance of algorithm 1. The structures given in figure 2 are initial programs for the well known recursive functions *last* (fig. 2a, tail recursive), *member* (fig. 2b, tail recursive with two conditional expressions), *addlist* (fig. 2c, linear recursive, similar to *clearblock*) and the function *myadd* (fig. 2d), which has a constant part not included in the recursive structure.

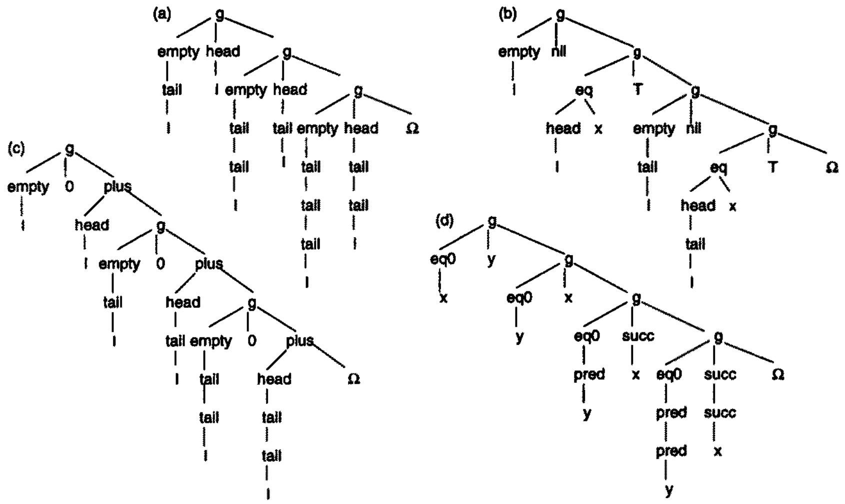

Fig. 2. Examples for recursive structures: tail recursion (a), tail recursion with two conditionals (b), linear recursion (c) and tail recursion with constant initial part (d)

Function *last* can be folded by the first hypothesis algorithm 1 generates. Function *member* can be folded by the second hypothesis, that tr consists of two conditional expression: $\mathcal{G} = g(empty(l), nil, g(eq(head(l), x), T, \mathcal{G}_{[tail(l)/l]}))$. Function *addlist* can be folded by the first hypothesis again, similar to the *clearblock*-problem given above. Function *myadd* can be folded after the hypotheses for tr starting at the root and consisting of one or two conditional expressions have failed by starting for a recursive structure in G at the second conditional expression. The RPS can be constructed by concatenating the constant part $g(eq0(x), y, \Omega)$ with $\mathcal{G} = g(eq0(y), x, \mathcal{G}_{[succ(x)/x, pred(y)/y]})$. (Note that the initial programs for *member* and *myadd* have to be expanded one level deeper to validate the hypothesis for σ).

We presented only cases for structures growing in the *else*-part of a conditional expression (i.e. right-recursive structures). But our algorithm can also deal with left-recursive cases and with structures with more than one recursion point as well. For example we can synthesize the function *maxlist* which consists of two tail recursions:

$$\mathcal{G} = g(empty(l), x, g(greater(head(l), x), \mathcal{G}_{[tail(l)/l, head(l)/x]}, \mathcal{G}_{[tail(l)/l]}))$$

and the tree recursive function for calculating *fibonacci*-numbers:

$$\mathcal{G} = g(eq0(x), 1, g(eq1(x), 1, plus(\mathcal{G}_{[pred(x)/x]}, \mathcal{G}_{[pred(pred(x))/x]}))).$$

5 Integrating Planning, Program Synthesis and Analogical Problem Solving

The algorithm for induction of RPSs is the core of the system IPAL which is implemented in a first prototype in LISP. In IPAL we realize our idea of combining inductive program synthesis with planning and analogical problem solving. We give a short description of both aspects in the following.

5.1 Generating Initial Programs by Planning

In separating the generation of initial programs from examples from program synthesis itself we are trying to model the behavior of a novice programmer or problem solver, who first tries to solve a given problem in a straight-forward way (i.e. constructing an initial program) and than generalizes his solution strategy. For generating initial programs from examples we use a backward-planner which is able to handle conjunctive top-level goals (and is able to deal with conflicting goals). The most important feature of our planner is, that it does not cover a single initial state only but a set of initial states in one planning trial. The planning algorithm is based on a technique proposed by [21] which has some similarities to approaches to universal planning (cf. [17]).

We are not presenting the planner here. Instead we illustrate the general idea with the *clearblock*-example introduced above, which has one top-level goal only. A novice or our system may be confronted with towers consisting of one, two or three blocks. These situations are initial states for our planner corresponding to the input part of examples in program synthesis. The initial states are described by conjunctions of predicates. Three possible initial states for the *clearblock*-problem are given in figure 3.

The system has to solve the problem to clear the base of the tower, i.e. block C. The goal state, which corresponds to the output part of examples in program synthesis, is represented by a single predicate: $cleartop(C)$. Note that our planner can deal also with conjunctions of goals as for example $on(A,B), on(B,C)$. The novice or our system has knowledge about operations and their conditions of application, corresponding to the background knowledge used in ILP approaches. This knowledge is represented as production rules with ADD and DEL lists for specifying the semantic of an operation. The operation needed to achieve the goal to clear a block is *on(x,y), cleartop(x)* $\rightarrow$ *puttable(x); ADD cleartop(y) DEL on(x,y).*

Our planner builds a tree starting with the goal as root node. A left branch is introduced for the case that the goal is already fulfilled and a right branch for the case that it is not. The right branch is labelled with that operation which is given at the righthand side of a production rule containing the desired predicate in the ADD-list. Predicates given in the condition part of this rule and not occuring in the DEL-list are introduced as new subgoals. That is, our planner works with a backward chaining algorithm. Variables occuring in the goal predicate are used to instantiate the selected production rules. If there remain unbound variables these are instantiated by a lookup in the set of initial states.

In our example we start with *cleartop*(C) as the root. In the left successor, no subgoal is left. Here we terminate with a leaf s which represents a situation variable. In the right branch we introduce the operator *puttable*(x). Variable y in the production rule is instantiated with C and we find situations s_1 and s_2 in our set of initial states so that $on(x, C)$ can be unified with $on(B, C)$ and variable x in the production rule is instantiated accordingly. A new subgoal, *cleartop*(B) is introduced as right successor-node. The resulting plan is given in figure 3.

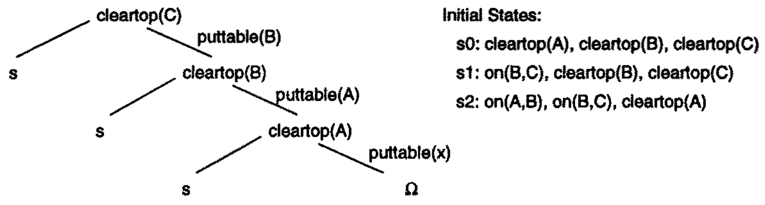
cleartop(C)
puttable(B)
s
cleartop(B)
puttable(A)
s
cleartop(A)
puttable(x)
s
Ω
Initial States:
s0: cleartop(A), cleartop(B), cleartop(C)
s1: on(B,C), cleartop(B), cleartop(C)
s2: on(A,B), on(B,C), cleartop(A)

Fig. 3. Plan for dealing with the clearblock-problem

The plan can be reformulated with help of the background knowledge $on(x, y) \equiv topof(y) = x$. Thereby we gain the initial program which is than used in program synthesis.

5.2 Programming by Analogy

If an RPS was succesfully generated from an initial tree, we store it in memory. These RPSs enlarge the set of predefined functions available to IPAL and they can be used for analogical programming. Analogical programming can be seen as a special case of analogical problem solving which is described by four subprocesses in cognitive science literature [15]: (1) Retrieval of an example problem already solved which is structural similar to the current problem; (2) mapping the structures of example and goal problem; (3) adapt the example solution to the goal problem; and (4) generalize over the structure of example and goal problem.

In the case of our programming problems, we assume, that a programmer/the system has already constructed an initial program and now wants to fold it to an RPS. The RPSs stored in memory can be expanded to initial programs of a given depth by the rewrite rule given in definition 6. We compare (map) the current initial program for which the RPS is unknown with initial programs expanded from RPSs stored in memory. Comparision of initial programs is done by a transformation algorithm: The current initial tree is tried to be made identical to an initial tree belonging to an RPS expanded from memory by substitution, deletion and insertion of nodes (i.e. symbols contained in the initial program term). This method is implemented as a special version of the tree-to-tree distance algorithm proposed by [12]. If the transformation can be performed by unique subsitutions of symbols only, the structures of the example and goal tree are isomorphic. The new RPS can than be gained by subsituting symbols of the example RPS according to the mapping function. We can show that this is less expensive than performing inductive program synthesis if the memory is effectively organized.

Currently we are employing this method of analogical problem solving for isomorphic structures only. But we are working on an extension to non-isomorphic cases, modifying example RPSs by deletion and insertion rules.

6 Conclusions and Future Research

We believe that segmenting inductive program synthesis in two parts - building initial programs by planning and folding initial programs to RPSs by a generic algorithm - may be a fruitful approach. Starting with an initial program as input makes program synthesis itself a not too difficult task and we can deal with complex recursive structures. The problem of inductive program synthesis from examples of course remains with all its well known difficulties. We only shift the greater part of the burden to the subproblem of generating initial programs from examples. But here we see possibilities to overcome some limitations in employing planning methodologies using heuristic techniques proposed in artificial intelligence. Additionally we look at our approach as a potential model for human skill acquisition in the domain of problem solving. We can describe skill acquisition by three levels of generalization, all reported in cognitive science literature: the chuncking of rules in learning by doing (i.e. building initial programs), descriptive generalization of the problem solving experience to a more general problem solving strategy (i.e. inductive program synthesis) and making use of already solved problems in analogical reasoning (i.e. giving an RPS a new semantic by interpreting the operation symbols w.r.t. another domain model).

Currently we can synthesize a variety of recursive structures from initial programs but our approach is restricted to independend variables, that is, we can not deal with nested loops. Our next goal therefore is, to extend the synthesis algorithm to finding dependencies in variable substitution. That is, we have to extend the "find subsitution" part of our algorithm from simple unification to a second cycle of induction. Another restriction to our approach is, that initial programs can be generated only for problems with finite data structures and primitive operations which can be defined by production rules, as blocksworld or Tower of Hanoi problems. While we are able to fold given initial programs representing numerical or list problems to RPSs, we are not able to construct initial programs from examples for these domains. To overcome this limitation we have to expand our planning algorithm from applying user-defined production rules only to making use of the operational semantics of already built-in functions of a given programming language (as *plus* or *tail*) and predicates (as *eq*0 or *empty*). That is, we have to integrate the eval-apply interpreter of a functional language (as for example LISP) in plan construction. Last but not least, our approach can be extended to use not only predefined functions but making use of already inferred RPSs which makes it possible to infer complex recursive programs containing subprograms.

Acknowledgements

We thank a number of students participating in our students projects for supporting the implementation of the IPAL system, especially: Mark Müller and Martin Mühlpfordt.

References

1. J.R. Anderson. Knowledge compilation: A general learning mechanism. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning - An AI Approach*, volume 2, pages 289–310. Tioga, 1986.
2. J.R. Anderson, P. Pirolli, and R. Farrell. Learning to program recursive functions. In M.T.H. Chi, R. Glaser, and M.J. Farr, editors, *The Nature of Expertise*, pages 153–183. Lawrence Erlbaum, 1988.
3. A. W. Biermann, G. Guiho, and Y. Kodratoff, editors. *Automatic Program Construction Techniques.* Collier Macmillan, 1984.
4. G. Le Blanc. BMWk revisited: Generalization and formalization of an algorithm for detecting recursive relations in term sequences. In F. Bergadano and L. de Raedt, editors, *Machine Learning, Proc. of ECML-94*, pages 183–197, 1994.
5. W. W. Cohen. Desiderata for generaization-to-n algorithms. In *Int. Workshop AII '92, Dagstuhl Castle, Germany*, volume LNAI 642, pages 140–150. Springer, 1992.
6. B. Courcelle and M. Nivat. The algebraic semantics of recursive program schemes. In Winkowski, editor, *Math. Foundations of Computer Science*, volume 64 of *LNCS*, pages 16–30. Springer, 1978.
7. J. Engelfriet. *Simple Program Schemes and Formal Languages.* Springer, 1974.
8. A.J. Field and P.G. Harrison. *Functional Progamming.* Addison-Wesley, 1988.
9. P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. to appear.
10. M. Ginsberg. Universal planning: An (almost) universally bad idea. *AI Magazine*, 10(4):40–44, 1989.
11. J. P. Jouannaud and Y. Kodratoff. Characterization of a class of functions synthesized from examples by a summers like method using a 'B.M.W.' matching technique. In *IJCAI*, pages 440–447, 1979.
12. S. Lu. A tree-to-tree distance and its application to cluster analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):219–224, 1979.
13. Z. Manna and R. Waldinger. How to clear a block: a theory of plans. *Journal of Automated Reasoning*, 3(4):343–378, 1987.
14. S. Muggleton. Learning from positive data. In S. Muggleton, editor, *Proc. of the 6th Int. Workshop on Inductive Logic Programming*, pages 225–244. Stockholm University, Royal Institute of Technology, 1996.
15. L. R. Novick and K. J. Holyoak. Mathematical problem solving by analogy. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 14:510–520, 1991.
16. U. Schmid and F. Wysotzki. Skill acquisition can be regarded as program synthesis. In U. Schmid, J. Krems, and F. Wysotzki, editors, *Proc. of the First European Workshop on Cognitive Modelling (TU Berlin)*, pages 39–45, 1996.
17. M.J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *IJCAI '87*, pages 1039–1046, 1987.
18. P. D. Summers. A methodology for LISP program construction from examples. *Journal ACM*, 24(1):162–175, 1977.
19. M. Veloso, J. Carbonell, M. A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The prodigy architecture. *J. of Experimental and Theoretical AI*, 7(1):81–120, 1995.
20. F. Wysotzki. Representation and induction of infinite concepts and recursive action sequences. In *Proc. of the 8th IJCAI, Karlsruhe*, 1983.
21. F. Wysotzki. Program synthesis by hierarchical planning. In P. Jorrand and V. Sgurev, editors, *Artificial Intelligence: Methodology, Systems, Applications*, pages 3–11. Elsevier Science, Amsterdam, 1987.