Applying Inductive Program Synthesis to Macro Learning

Ute Schmid and **Fritz Wysotzki**
Dept. of Computer Science, Technical University Berlin
Franklinstr. 28, D-10587 Berlin, Germany
{schmid, wysotzki}@cs.tu-berlin.de

Abstract

The goal of this paper is to demonstrate that inductive program synthesis can be applied to learning macro-operators from planning experience. We define macros as recursive program schemes (RPSs). An RPS represents the complete subgoal structure of a given problem domain with arbitrary complexity (e. g., *rocket* transportation problem with n objects), that is, it represents domain specific control knowledge. We propose the following steps for macro learning: (1) Exploring a problem domain with small complexity (e. g., *rocket* with 3 objects) using an universal planning technique, (2) transforming the universal plan into a finite program, and (3) generalizing this program into an RPS.

Introduction

Interest in learning macro-operators for planning (Minton 1985; Korf 1985) has decreased over the last decade, mainly because of the utility problem (Minton 1985). But new results in reinforcement learning are promising – showing that more complex problems are solvable and that planning can be speed-up considerably when applying macros (Precup & Sutton 1998). In classical planning, learning is currently investigated mainly in the context of the acquisition of domain specific control knowledge (Borrajo & Veloso 1996). We consider *recursive* macros as a special case of control knowledge. Classical linear macros restrict search by offering sequences of operators which can be applied instead of primitive actions. Iterative or recursive macros (Shell & Carbonell 1989; Shavlik 1990) ideally eliminate search completely by representing the complete subgoal structure (solution *strategy*) of a problem domain. For example, a macro for a one-way transportation problem as *rocket* (Veloso & Carbonell 1993) represents the strategy that *all* objects has to be loaded before the rocket moves to its destination.

We propose to apply a technique of inductive program synthesis (Summers 1977; Wysotzki 1983; Schmid & Wysotzki 1998) to macro learning. Inductive program synthesis algorithms learn recursive programs from a small set of input/output examples. Learning is performed by a two-step process: in a first step, I/O examples are transformed into a finite program; in a second step, the finite program is generalized to a recursive program. This second step is also called generalization-to-n and corresponds to programming by demonstration (Cohen 1998). While the mutual benefit of combining planning and program synthesis is recognized in the deductive field (Manna & Waldinger 1987), there is only limited cross-fertilization of planning and *inductive* program synthesis: Shavlik (1990) demonstrates how explanation-based learning can be applied to learning recursive concepts from problem solving examples; Shell & Carbonell (1989) show analytically and empirically how iterative macros can reduce planning effort and point out that learning iterative macros has to rely on generalization-to-n algorithms; Kalmar & Szepesvari (1999) discuss learning and efficiency of iterative macros in the context of Markov decision problems.

Our overall approach to learning recursive macros from some initial planning experience consists of three steps: First, a problem of small complexity is explored by universal planning. For example, a plan for the *rocket* one-way transportation problem is generated for three objects. A plan cannot be generalized directly – it has first to be transformed into a finite program, that is, a conditional expression giving the action sequences for transforming different states into a state fulfilling the desired goals. This finite program is generalized to a recursive macro, e. g., a macro solving *rocket* problems for an arbitrary number of objects. In the following, we will describe these steps – plan construction, plan transformation, and generalization – in detail.

Optimal Universal Plans

Our planning system DPlan is designed as a tool to support the first step of inductive program synthesis – generating finite program traces for transforming input examples into the desired output. Because our work is in the context of program synthesis, planning is for deterministic and (small) finite domains only and completeness and optimality are of more concern than efficiency considerations. DPlan is a state-based, non-linear, total-order backward planner. DPlan is similar to universal planning (Schoppers

Table 1: The *Rocket* Domain

$\mathcal{D}$ = { ((at O1 B) (at O2 B) (at Rocket B)),
((inside O1 Rocket) (at O2 B) (at Rocket B)),
((inside O2 Rocket) (at O1 B) (at Rocket B)),
((inside O1 Rocket) (inside O2 Rocket) (at Rocket B))
((inside O1 Rocket) (inside O2 Rocket) (at Rocket A)),
((at O1 A) (inside O2 Rocket) (at Rocket A)),
((at O2 A) (inside O1 Rocket) (at Rocket A)),
((at O1 A) (at O2 A) (at Rocket A)) }

$\mathcal{G}$ = {(at O1 B), (at O2 B)}
$\mathcal{O}$ = {load, unload, move-rocket} with

load(o l)	**move-rocket**
PRE {(at o l), (at Rocket l)}	**PRE** {(at Rocket A)}
ADD {(inside o Rocket)}	**ADD** {(at Rocket B)}
DEL {(at o l)}	**DEL** {(at Rocket A)}

unload(o l)
PRE {(inside o Rocket), (at Rocket l)}
ADD {(at o l)}
DEL {(inside o Rocket)}

1987) and conditional planning (Peot & Smith 1992; Borrajo & Veloso 1996): instead of a plan representing a sequence of actions transforming a *single* initial state into a state fulfilling the top-level goals, DPlan constructs a planning tree, representing optimal action sequences for *all* states belonging to the planning domain. A planning tree represents the same information as a state-action table (Schoppers 1987) but in a more compact way. Plan construction is based on breadth-first search and therefore works without backtracking.

The general idea of DPlan is to construct a plan which represents a minimal spanning tree (Christofides 1975) of the (implicitly) given state space with the goal state(s) as root. In the current implementation, we present the complete set of states $\mathcal{D}$ as input. Planning problems are defined in the following way:

Definition 1 (Planning Problem) *A planning problem $\mathcal{P}(\mathcal{O}, \mathcal{D}, \mathcal{G})$ consists of a set of operators $\mathcal{O}$, a set of problem states $\mathcal{D}$, and a set of top-level goals $\mathcal{G}$. Operators are defined with preconditions φ and effects. Effects are given by ADD- and DEL-lists (A, D). Conditioned effects are given by effect-preconditions φ_i and effects A_i, D_i. Currently, preconditions, effects and goals are restricted to conjunctions of positive literals and variables are existential quantified only. In contrast to other planners, domain $\mathcal{D}$ is not the set of constants occuring in a domain but the set of all possible states. A state is a conjunction of atoms (instantiated positive literals).*

An example for the specification of a planning problem from the *rocket* domain (Veloso & Carbonell 1993) is given in table 1. The planning goal is to transport two objects $O1$ and $O2$ from a place A to a destination B. The transport vehicle (rocket) can only be moved in one direction (A to B). Therefore, it is important to load *all* objects before the rocket moves to its destination. The resulting planning tree is given in figure 1.

Plan *construction* is done by backward-operator application. A plan can be *executed* by forward application of all operators on the solution path from some initial state towards the root of the plan. Operator application is defined for fully-instantiated operators o (actions). In the case of operators with conditioned effects an instantiated operator is constructed for each possible effect. As usual, operator definitions are given in a set-theoretical way:

Table 2: DPlan algorithm

- Input: a planning problem $\mathcal{P}(\mathcal{O}, \mathcal{D}, \mathcal{G})$
- Output: minimal spanning tree Θ for $\mathcal{P}$, or $\perp$

1. Initialization (root of Θ):
   Let $S_\mathcal{G} = \{S_i \mid i = 0 \ldots n; S_i \in \mathcal{D} \text{ and } \mathcal{G} \subseteq S_i\}$
   - if $S_\mathcal{G} = \emptyset$: $\Theta = \perp$
   - if $\|S_\mathcal{G}\| = 1$: $\Theta = S$ for $S_\mathcal{G} = \{S\}$, $\mathcal{D} = \mathcal{D} \setminus S_\mathcal{G}$
   - else (more than one state fulfilling the top-level goals): $\Theta = \mathcal{G}(S_1 \ldots S_n)$ (a term/tree with the top-level goals as root and all states fulfilling $\mathcal{G}$ as children) for $S_\mathcal{G} = \{S_1, \ldots, S_n\}$, $\mathcal{D} = \mathcal{D} \setminus S_\mathcal{G}$
2. Tree expansion (if $\Theta \neq \perp$):
   For all leafs S_i of Θ: calculate all predecessors S'_{ij} with $Res^{-1}(S_i, o_{ij}) = S'_{ij}$ and $S'_{ij} \in \mathcal{D}$ where o_{ij} is an instantiated operator from $\mathcal{O}$
   $\Theta(S_i) = S_i(o_{i1}(S'_{i1}) \ldots o_{in}(S'_{in}))$ (a sub-term/sub-tree where node S_i has children $S'_{i1} \ldots S'_{in}$ with the connecting arcs labeled with actions $o_{i1} \ldots o_{in}$), $\mathcal{D} = \mathcal{D} \setminus \{S'_{ij}\}$
3. Termination: $\mathcal{D} = \emptyset$ or no leaf in Θ can be expanded

Definition 2 (Operator Application) *Forward application of an instantiated operator (action) is defined as $Res(S, o) = S \setminus D \cup A$ if $\varphi \subseteq S$; backward application as $Res^{-1}(S, o) = S \setminus A \cup \{D \cup \varphi\}$ if $A \subseteq S$. With $Res_p^{-1}(S, \{o_1 \ldots o_n\})$ we represent "parallel" application of the set of all actions which fulfill the application condition for state S resulting in a set of predecessor states $\{S'_1 \ldots S'_n\}$.*

Operator application does not include an admissibility check. That is, whether an action generates a valid successor or predecessor state has to be determined by a higher instance, namely the planning algorithm. Note, that subtraction and union of sets are only commutative if these sets are disjoint.

The DPlan algorithm (see tab. 2) is given here in an abbreviated version and we will describe it informally. Full formalization, proofs of termination, soundness, and optimality are given in Schmid (1999). The algorithm works in the following way: First, it is checked whether there is at least one state in the set of all states $\mathcal{D}$ which fulfills the top-level goals. If no state S with $\mathcal{G} \subseteq S$ exists, planning terminates without success. If one such state exists, this state is introduced as root of the planning tree. If more than one state in $\mathcal{D}$ fulfills $\mathcal{G}$, $\mathcal{G}$ is introduced as root node and all states fulfilling $\mathcal{G}$ are introduced as children with unlabeled arcs (representing "empty" actions). All states introduced in the planning tree are removed from $\mathcal{D}$.

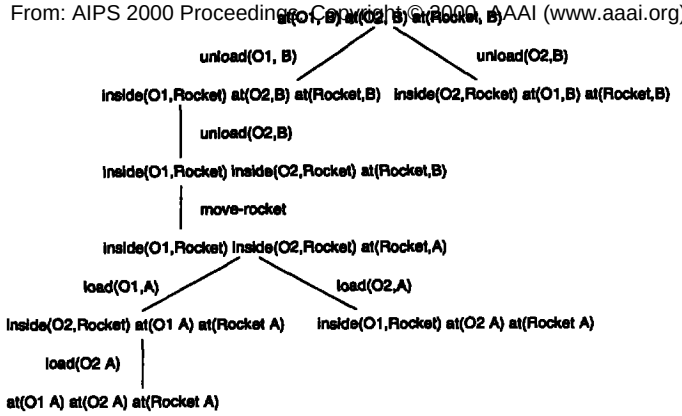


Figure 1: A Planning Tree for the *Rocket Problem*

After constructing the root of the planning tree, DPlan proceeds recursively for each leaf calculating all immediate predecessor states. The predecessors are identified by backward operator application $Res_p^{-1}(S, \{o_1 \ldots o_n\})$ where $\{o_1 \ldots o_n\}$ is the set of all instantiated operators fulfilling the application condition for backward application (i. e., the Add-List of o_i matches a subset of S), and $Res_p^{-1}(S, \{o_1 \ldots o_n\})$ returns all predecessors of S. For each admissible predecessor S'_{ij}, with $o(S'_{ij}) = S_i$, o is introduced as arc and S'_{ij} as child node.

The state set $\mathcal{D}$ is used for calculating predecessors in the following way: Free variables in an operator are instantiated only in accordance with the states in $\mathcal{D}$ – that is, the number of action candidates is restricted; only predecessors which are in $\mathcal{D}$ are accepted as admissible – that is, inconsistent states (which never were members of $\mathcal{D}$) as well as states which are already included in the planning tree (already removed from $\mathcal{D}$) are omitted. This is the reason why the leaf node of the upmost right-hand branch in figure 1 is not expanded – application of *unload(O1,B)* to that node results in a predecessor state *(inside O1 Rocket) (inside O2 Rocket) (at Rocket B)* which is already covered in the left-hand branch of the plan. Because $\mathcal{D}$ is reduced in each planning step, planning is linear in the number of states.

The algorithm terminates successfully with a minimal spanning tree if $\mathcal{D}$ is empty. If $\mathcal{D}$ is not empty and nevertheless there is no operator applicable which leads to a state in the set of remaining states $\mathcal{D}$, the graph (problem space) underlying the domain $\mathcal{D}$ might be disconnected – that is, there are some problem states from which the goal cannot be reached. This is for example the case in the *monkey* domain: when the monkey climbs onto the box and he is not at the position of the bananas he has reached a dead-end, because the domain does not provide an "unclimb" operator.

DPlan constructs optimal action sequences: For each state in a planning tree the path starting at this node and ending at the root gives the shortest possible sequence of actions transforming this state into the goal. Note, that in general the minimal spanning tree of a graph is not unique. For the *rocket* problem (see fig. 1) exist four minimal spanning trees (dealing first with $O1$ vs. with $O2$ when loading and unloading).

Alternatively to DPlan, each existing planning system (e. g., *Prodigy*, Veloso *et al.* 1995) can be used to construct minimal spanning trees by replacing the incorporated planning strategy (selection of one action for each planning step) by an "exploration mode" where all applicable actions are stored. Remember, that this kind of universal planning is performed for *learning* recursive macros from some initial experience when solving a problem of small complexity. Human problem solvers also must invest much mental effort if they want to extract a general solution strategy – for example for solving the Tower of Hanoi problem (Klahr 1978). Macro *application* can be performed in the context of standard problem solving or planning strategies – reducing the effort by applying such "compiled" knowledge.

Planning with Macros

Introducing macro-operators into planning has two advantages: the number of match-select-apply cycles gets reduced and macros guide the sequence in which planning goals are attacked. On the other hand, introduction of macros might involve the utility problem: If a system gets "swamped" with macros, the number of operator candidates which have to be matched with the current state increases. One strategy to reduce this problem is to estimate the utility of a potential macro and only store selected macros (Minton 1985). An alternative strategy is, to learn more general macros, for example iterative or recursive (for short "cyclic") macros (Shell & Carbonell 1989). For example, there might be a macro *load-all* which fulfills the goal {*(inside* ob_1 *Rocket), ... (inside* ob_n *Rocket)*} for all objects at some given place. In the case of linear macros – i. e., operator compositions – loading all objects might involve either learning $n-1$ separate macros (for loading 2, or 3, ... up to n objects) or multiple macro-calls (e.g., $n/2$ calls of a *load-two-objects* macro).

While for a linear macro its preconditions must be checked to decide whether it is applicable, a cyclic macro can be indexed by the goal it promises to make true (Shavlik 1990). For example, if there are pending goals {*(inside O1 Rocket) (inside O2 Rocket)*} the *load-all* macro is activated. Macro-application is encapsulated (Shavlik 1990). That means, that until the macro terminates no other operators are checked for applicability. Thereby the number of match-select-apply cycles gets reduced by k if the *load* operator has to be applied k times. When integrating macro-application into plan construction optimality cannot be guaranteed (c. f., Kalmar & Szepesvari 1999): A recursive macro generates new goals for each application (e. g., loading of the next object) which cannot be interleaved with other pending goals (e. g., moving the rocket).

If there exists knowledge about the structure of a planning domain, this knowledge can be incorporated into the domain specification by pre-defining control

rules, for example as recursive macros. The more interesting case is that such macros are learned from some initial planning experience. In the following sections, we will describe, how recursive macro-operators can be learned by exploring a problem of small complexity. For example, we can generate a plan for unloading three objects and generalize to *unload-all*. This is done by first transforming the initial plan into a finite program and than applying a generalization-to-n algorithm.

We realize two strategies for macro learning also proposed in reinforcement learning (Sun & Sessions 1999): (1) incremental elemental-to-composite learning and (2) simultaneous composite learning. In the first case, the system is initially trained with simple tasks and the learned macros afterwards can be used when solving/learning a complex task – learning macros which contain other macros. In the second case, the system immediately learns the complex task and has to perform the decomposition autonomously (Wysotzki 1983). For the *rocket* problem, application of the first strategy means that we first learn the *load-all* and *unload-all* macros and use them in *rocket*. Application of the second strategy means, that we identify sub-plans in *rocket*, generalize them separately, and replace these sub-plans by the macro-name.

In the classical functional approach to inductive program synthesis (Summers 1977; Wysotzki 1983; Le Blanc 1994) the generation of recursive programs from I/O examples is separated into two distinct steps: Rewriting a set of I/O examples into a finite program and generalization-to-n. The first step corresponds roughly to the concept of "saturation" in inductive logic programming (Flener & Yilmaz 1999). The result – i. e., the form and complexity of the finite program – is completely dependent on the background knowledge (here: predefined functions and predicates) provided for the rewrite-system. Additionally, the outcome depends on the used rewrite-strategy – i. e., even for a constant set of background knowledge rewriting can result in different programs. Theoretically, there are infinitely many possible ways to represent a finite program which describes how input examples can be transformed in the desired output. Because generalizability depends on the form of the finite program, this first step is the bottleneck of program synthesis. Here program synthesis is confronted with the crucial problem of AI and cognitive science – problem solving success is determined by the constructed representation (Kaplan & Simon 1990).

We propose to use planning to realize rewriting of I/O examples. Inputs correspond to problem states, outputs to states fulfilling the desired goals, and transformation from input to output is realized by calculating the optimal action sequence. The resulting minimal spanning tree already represents the structure of the searched-for program. Nevertheless, the plan cannot be generalized directly, but some transformation steps are needed to generate a finite program which than can be input to a generalization-to-n algorithm.

Transforming Plans into Programs

In the following, we describe how a universal plan can be transformed into a finite program which can be input to a generalization-to-n algorithm. The basic concepts of our transformation strategy are adopted from program synthesis: We will interpret plans in situation calculus (Manna & Waldinger 1987) and we will provide knowledge about the data structures of a domain (Manna & Waldinger 1987; Summers 1977). Transformation involves the following steps: (1) Inferring the data structure underlying the problem from the structure of the planning tree and replacing constants by constructive expressions, (2) minimizing the planning tree by deletion of than identical parts and/or identification of sub-plans, and (3) introducing conditional expressions. For all transformation steps it has to be guaranteed that the semantic of the original plan is preserved. We will see, that the first step is the bottleneck of plan transformation.

We interpret the plan as finite program in the following way (see also Wysotzki 1987):

Definition 3 (Plan as Program) *Each node S (set of literals) in plan Θ is interpreted as conjunction of boolean expressions B. The planning tree can now interpreted as nested conditional: IF $B(s)$ THEN s ELSE $o(\Theta'(s))$ with s as situation variable, o as action given at the arc from B to its child node, and Θ' as sub-plan with this child node as root. (Read: "*if B holds in the current situation then return it else apply action o to a situation where the literals given in the next node are fulfilled*").*

This "program", however, is not fit for generalization. The main reason is, that actions are applied to constants (e. g., objects *O1, O2* in fig. 1) which belong to an unordered set.

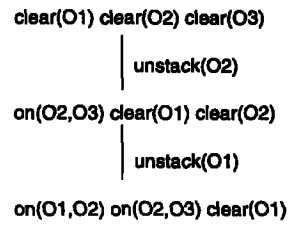
A crucial pre-requisite for program synthesis is knowledge about the *data structures* of the domain (Manna & Waldinger 1975). For synthesizing a program over lists (as *reverse* or *append*), knowledge about construction of lists has to be provided. Summers (1977), for example, uses a predefined ordering over lists together with the selector functions $head(l)$ and $tail(l)$ as background knowledge. Manna & Waldinger (1987) introduce a *hat*-axiom for synthesizing a recursive program for clearing an arbitrary block: *if not clear(x,s) then on(hat(x,s),x,s)*. That is, they introduce a function *hat* returning that block lying on a block x together with a data structure $x < hat(x) < hat(hat(x)) \ldots$. In general, each problem relies on one or more of the following three underlying data structures:

- Linear sequence: $bot < next(bot) < next(next(bot)) < \ldots$, constituting a total order (each element has exactly one successor). Examples are natural numbers (0, $succ^i(0)$) or stacks of objects ($base$, $hat^i(base)$). Besides the constructor $next$ a test for bot (e. g., $= 0$?) is needed.
- List: with bottom element nil and constructor $cons(e, list)$, constituting a complete partial order. For this composed data type exist an empty-test ($null$?) and two selectors $head(cons(e, list)) = e$ and $tail(cons(e, list)) = list$.

$\mathcal{D}$ = { ((clear O1) (clear O2) (clear O3)) ,
((on O2 O3) (clear O1) (clear O2)),
((on O1 O2) (on O2 O3) (clear O1)) }

$\mathcal{G}$ = {(clear O3)}
$\mathcal{O}$ = {unstack} with

unstack(x)
PRE {(clear x), (on x y)}
ADD {(clear y)}
DEL {(on x y)}



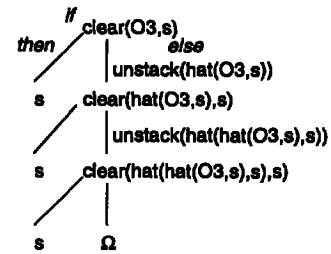


Figure 2: The *Unstack* Problem, its Linear Plan, and the Resulting Finite Program

- Set: with bottom element $\emptyset$ and constructor $put(e, set)$, constituting a lattice (with the set of all elements as top-element). For this composed data type exist an empty-test ($empty?$) and two selectors $pick(set) = e, e \in set$ and $rest(set, e) = set \setminus e$.

Which of these data structures is underlying a given problem can be inferred from the structure of the minimal spanning tree. If a plan is linear (without branching), the underlying data structure is assumed to be either a linear sequence or a list, otherwise a list or a set. If there exist more than one minimal spanning tree for a problem, this indicates that the sequence in which constants are dealt with is irrelevant, that is, that the underlying data structure is a set. For example, the classical *tower* problem (Shavlik 1990) relies on a list (blocks are ordered by the goal after planning), the *rocket* problem relies on a set of objects.

Let us first consider a simpler domain than *rocket*: Assume that we have a stack of objects in the rocket and want to unload one of these objects. To unload it, it must be on top of the stack, that is, some objects might have to be unstacked first. For the problem given in figure 2 we want to unload object $O3$. The resulting minimal spanning tree is linear and gives us an ordering of objects: $O3 < O2 < O1$. That is, we can introduce a linear data structure with $bot = O3$, $next(O3) = O2$ and $next(next(O3)) = O1$. Because the macro to be learned has to be applied to the state descriptions (sets of literals) of a domain, we have to provide knowledge how *next* can be determined from these descriptions. For our example holds $on(x, y) \equiv on(next(y), y)$, that is *next* corresponds to the *hat*-axiom introduced above. Preferably, we would like to learn this association. But currently we provide this knowledge as domain axioms. That is, after the data structure is inferred, we check whether the constructor- and selector functions correspond with predefined domain axioms.

Now the plan can be rewritten recursively by standard term rewriting (Dershowitz & Jouanaud 1990) replacing the constants by the constructive expression associated with them. For planning *trees*, rewriting is performed for each linear sub-plan, because the different branches of the plan might involve different bottom-elements. At the end of this first transformation step, all literals whose arguments are not involved in rewriting are removed from each state node in the plan. Because the minimal spanning tree represents the order in which goals (top level and introduced as operator preconditions) are realized, the remaining literals correspond to the goals fulfilled during planning and all other literals are considered as irrelevant for generalization. Because the plan is already linear, the second step of transformation – minimization of the tree – can be skipped. The resulting finite program is given in figure 2: the remaining literals (here only $clear(o, s)$ with $o \in \{O3, hat(O3, s), hat(hat(O3, s), s)\}$) are interpreted as boolean expressions and the plan is interpreted as nested conditional in accordance to definition 3, where the symbol Ω (undefined) is introduced at the end of each branch of the planning tree. Note, that because the plan is re-interpreted in situation calculus, each predicate and operation symbol is extended by one argument s, referring to the current situation. The term representation of the finite program is:

if clear(O3,s) then s else unstack(hat(O3,s),
if clear(hat(O3,s),s) then s else unstack(hat(hat(O3,s),s),
if clear(hat(hat(O3,s),s),s) then s else Ω)).

For program synthesis, we consider the remaining constants as variables.

Let us consider a more complex problem – unloading all objects from the rocket at some place. *Unloading* is a sub-domain of *rocket* or any other transportation domain. Objects can be inside a vehicle (*Rocket*) or already unloaded at the destination (*Place*); the goal is {*at(O1,Place),at(O2,Place),at(O3,Place)*}; and there is only one operator – *unload* (see tab. 1). The constructed plan given in figure 3 is one of 6 possible minimal spanning trees for this problem.

From the structure of the planning tree we can infer that the underlying data structure is a set of objects. The result of the first transformation step is given in figure 4 (with an abstract, more readable tree on the right-hand side). Note, that the predicate symbol $at^*(oset, Place)$ is now a generalization of the original $at(o, Place)$– taking sets of objects instead of single objects as argument. It corresponds to the $empty?$-test associated with *set* and is defined as $\forall(o)(at(o, Place) \wedge at(Rocket, Place) \equiv at^*(put(o, oset), Place))$, where *put* is the *set*-constructor. Selection of the object to be unloaded is realized by the selector *pick*, the remaining objects are represented by the *rest* selector.

This step of plan transformation results in re-

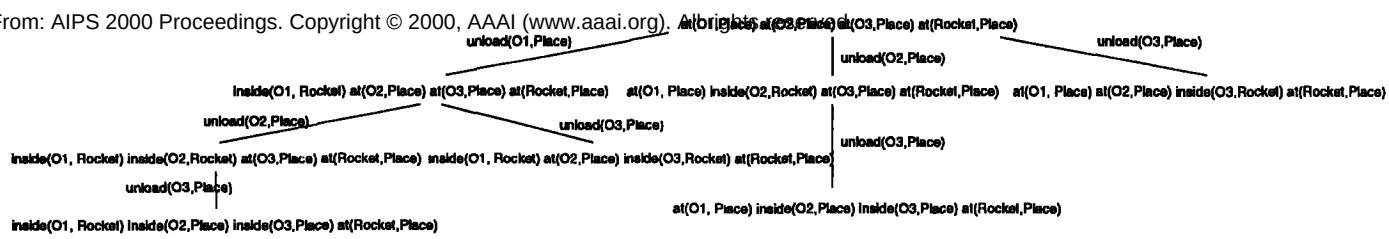


Figure 3: Planning Tree for the *Unloading* Problem (one of 6 possible minimal spanning trees)

ducing the planning tree into a linear sequence because all branches are structurally equivalent (anti-unifiable,Plotkin 1970). The resulting program is:

if at(oset,Place,s) then s else unload(pick(oset),*
if at(rest(oset,pick(oset)),Place,s) then s*
else unload(pick(rest(oset,pick(oset))),
if at(rest(rest(oset,pick(oset)),pick(oset)),Place,s) then s*
else unload(pick(rest(rest(oset,pick(oset)),pick(oset))),
if at(rest(rest(rest(oset,pick(oset)),pick(oset)),pick(oset))),*
Place,s) then s else Ω))).

The variable *oset* represents one possible constellation of up to three objects *inside* the rocket: $oset \in \mathcal{P}(\{O1, O2, O3\})$. If no object is inside the rocket, $at^*(\{O1, O2, O3\}, Place, s)$ holds, if all three objects are inside the rocket, $at^*(nil, Place, s)$ holds. Depending on the sequence in which objects are selected by *pick*, one of the 3! possible *unload* sequences is realized.

We can generate an isomorph finite program from a plan for loading all objects. For the *rocket* plan given in figure 1, plan transformation results in a finite program composed of the *unload-all* and *load-all* programs, separated by the *move-rocket* operator. The second transformation step additionally involves identification of sub-plans: A sub-plan is defined as a segment of the planning tree which consists of structurally identical action-sequences (e. g., a sequence of *unloads*). These sub-plans are deleted from the plan, generalized separately, and labelled by the name of the generalized-macro. In general, the same macro can occur at different parts of the plan.

Generalizing Finite Programs

Now we want to describe the second step of program synthesis – generalization-to-n. In contrast to the classical approach and to inductive logic programming, our generalization-to-n algorithm is not restricted to a given programming language (Prolog or Lisp). Instead, we regard programs as elements of some arbitrary term algebra. That is, we synthesize recursive program *schemes* (RPSs) (Courcelle & Nivat 1978; Wysotzki 1983) and thereby we can deal with list and number problems which are typically considered in program synthesis in the same way as with planning problems (blocks-world, puzzles, transportation problems). An RPS can be mapped to a recursive program in some programming language by interpreting the symbols in accordance to the specification of this language.

Input in the generalization algorithm is a finite program term which is element of some term algebra:

Definition 4 (Finite Program) *A finite program is a term $t_S \in M(V, F \cup \Omega)$. M is a term algebra over variables V and function symbols F with Ω as the undefined (empty) term.*

Let us consider the finite program for un-stacking an object constructed above with $M(\{o,s\},\{clear, hat, unstack, if\text{-}then\text{-}else, \Omega\})$. The term can be valuated and interpreted in the usual way: o can be instantiated by (the name of) an object, s is a situation variable. Conditioned expressions are represented as *if x then y else z*. The predicate $clear(o, s)$ is realized by a boolean function which is true if $clear(o, s)$ holds in situation s and false otherwise. The operator $unstack(o, s)$ corresponds to the operator defined in figure 2, rewritten for situation calculus (Manna & Waldinger 1987): instead of being applied to the current state in working memory, it is applied on the set of literals with which variable s is instantiated. The selector function *hat* was defined above.

Output of the generalization algorithm is a recursive program scheme:

Definition 5 (Recursive Program Scheme) *An RPS is a pair $\langle \Sigma, t \rangle$ with $\Sigma = \langle T_i(v_1 \ldots v_n) = t_i \mid i = 1 \ldots n \rangle$ is a system of equations ("subroutines") and $t \in M$ is the "main program"; t_i and t are elements of the extended term algebra $M(V, F \cup \Phi)$ with V as set of variables, F as set of function symbols (with $f^i \in F$ we denote functions with arity i), and Φ as set of function variables (names of user defined functions); $T_i \in \Phi$, and $v_1 \ldots v_n \in V$.*

If T_i is contained in t_i, the equation defines a recursive function; t_i can also contain further T_j's, that is, make use of other functions.

An RPS generalizing the *unstack* term given above is $\Sigma = \langle$ *rec-unstack(o,s) = if clear(o,s), s, unstack(hat(o,s), rec-unstack(hat(o,s), s))* $\rangle$ with $t =$ *unstack-rec(o,s)* for some constant o and some set of literals s.

For program synthesis we reverse the idea of determining the semantic of a recursive function as its smallest fix-point (Wysotzki 1983; Schmid & Wysotzki 1998)[1]: from a given sequence of unfoldings we want to extrapolate the minimal recursive program scheme

[1]For an introduction to fix-point semantics, see (Field & Harrison 1988).

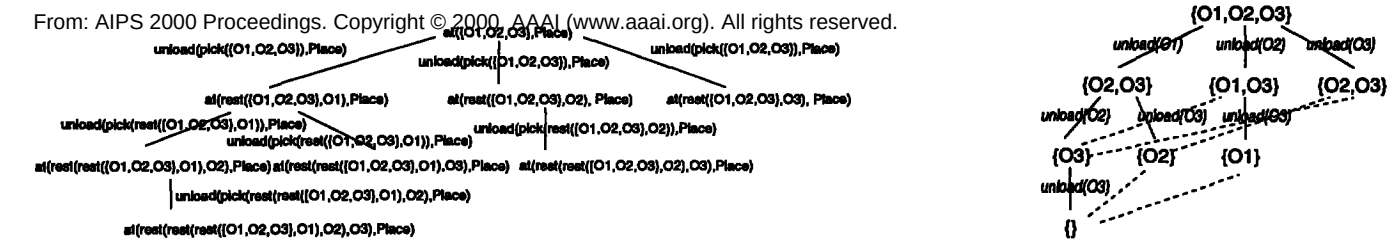


Figure 4: Introduction of Data Structure *Set* in the *Unloading* Plan

which can generate these unfoldings.[2]

Definition 6 (Folding of a finite program) *A finite program t_S can be folded into a recursive program scheme* iff *it can be decomposed into a sequence $\mathcal{T}^{(0)} = \Omega$, $\mathcal{T}^{(l)} = tr(\mathcal{T}^{(l-1)}_{[t/v]}/m)$ with $l = 1 \ldots n$, $\mathcal{T}^{(n)} = t_S$ of partial transformations which successively cover a larger amount of inputs.*

For our example we have:

$$
\begin{aligned}
\mathcal{T}^{(0)} &= \Omega \\
\mathcal{T}^{(1)} &= \mathit{if\ clear}(o,s)\ \mathit{then}\ s\ \mathit{else}\ \mathit{unstack}(\mathit{hat}(o,s),\Omega) \\
\mathcal{T}^{(2)} &= \mathit{if\ clear}(o,s)\ \mathit{then}\ s\ \mathit{else}\ \mathit{unstack}(\mathit{hat}(os), \\
&\quad \mathit{if\ clear}(\mathit{hat}(o,s),s)\ \mathit{then}\ s\ \mathit{else}\ \mathit{unstack}(\mathit{hat}(\mathit{hat}(o,s),s),\Omega)) \\
\mathcal{T}^{(3)} &= \mathit{if\ clear}(o,s)\ \mathit{then}\ s\ \mathit{else}\ \mathit{unstack}(\mathit{hat}(o,s), \\
&\quad \mathit{if\ clear}(\mathit{hat}(o,s),s)\ \mathit{then}\ s\ \mathit{else}\ \mathit{unstack}(\mathit{hat}(\mathit{hat}(o,s),s), \\
&\quad\quad \mathit{if\ clear}(\mathit{hat}(\mathit{hat}(o,s),s),s)\ \mathit{then}\ s\ \mathit{else} \\
&\quad\quad\quad \mathit{unstack}(\mathit{hat}(\mathit{hat}(\mathit{hat}(o,s),s),s),\Omega))) = t_S
\end{aligned}
$$

with tr = *if clear(o,s) then s else unstack(hat(o,s), m)* and substitution $[hat(o,s)\ /o]$. Because $\mathcal{T}^{(l)}$ = *if clear(o,s) then s else unstack(hat(o,s),* $\mathcal{T}^{(l-1)}_{[hat(o,s)/o]}$*)* holds for all T's we can fold t_S and obtain the RPS given above.

With our method we can infer tail recursive structures (for-loops), linear recursive structures (while-loops), tree-recursive structures and combinations thereof. For details about the formal background, the synthesis algorithm, its scope and complexity, see (Schmid & Wysotzki 1998).

Recursive macros for the *rocket* domain are:

unload-all(oset,l,s) = if at(oset,l,s) then s else*
unload(pick(oset),l,unload-all(rest(oset,pick(oset)),l,s))
load-all(oset,v,s) = if inside(oset,v,s) then s else*
load(pick(oset),v,load-all(rest(oset,pick(oset)),v,s))
rocket(oset,l,v,s) = unload-all(oset,move-rocket(l,s),
load-all(oset,l,v,s)).

The *load-all* macro assumes an infinite capacity of the vehicle. To take capacity restrictions into account, these information has to give to the planning system (*max-capacity(x)*) and we must include *current-load(x)* as resource variable (Koehler 1998).

[2]The idea of reversing a deductive approach for inductive inference is also used in inductive logic programming with the concept of inverse resolution (Flener & Yilmaz 1999).

Conclusions and Further Work

We demonstrated that planning can be combined with inductive program synthesis for learning recursive macro-operators. While we use DPlan for construction of planning trees, the same information can be obtained from other planning systems by storing the complete search tree. Because information about the structure of a domain is needed for generalization-to-n, we cannot use an incremental learning strategy as *Prodigy* (Veloso *et al.* 1995). Instead, a problem of small complexity is investigated completely, and this experience is used for generalization. This strategy mimics human problem solvers, for example in the field of program construction (Schmid & Wysotzki 1998): if one wants to compose a program for sorting lists, one first constructs some initial examples by hand-simulation, tries to identify their common structure and than generalizes over them. Construction of minimal spanning trees and generalization-to-n are completely implemented and formalized. For plan transformation we have a rudimentary implementation which works for plans with linear structures and for selected list and set problems. To extend our algorithm, we adopt a similar strategy as researchers in the program transformation domain (Smith 1990): we analyze as many problems as possible and try to extract heuristics covering problem classes which are as large as possible.

The focus of this paper was to introduce our approach to recursive macro learning and illustrate it with some examples. We did not present empirical results showing how recursive macros speed-up planning. Analytical results for efficiency gains when using cyclic macros are given by Shell & Carbonell(1989) and Kalmar & Szepesvari(1999). If there is already a macro for solving a problem, planning can be omitted completely. The costs are identical to execution of the macro (e. g. $2n+1$, i. e. a linear effort, for the *rocket* problem with n objects). If there is a macro for solving a sub-problem of a given problem, planning effort is reduced for these sub-problems: For example, if a pending goal is to clear some object in an n-object world, execution of the *unstack* macro has linear effort and up to n match-select-apply cycles together with a possible backtrack for each cycle can be omitted.

The costs for learning a recursive macro are necessarily high: first a problem has to explored by planning, than the planning tree has to be transformed into a finite program, and finally the finite program has to be

generalized. Planning effort is linear in the number of states – which can be already high for three-object problems, e. g. 27 for the *Tower of Hanoi* problem with three discs. Generalization over finite programs has exponential effort in the worst case (the problem cannot be generalized and all hypotheses for folding have to be generated and tested). Effort of plan transformation depends on the complexity of the provided background knowledge and the incorporated strategies. But because we start program construction not from the scratch – as typical in inductive program synthesis and genetic programming (Koza 1992) – but using planning together with the knowledge about legal operators as guideline, we can keep the amount of background knowledge and the effort of search for a generalizable program comparatively low. While we use only knowledge about data structures, Shavlik(1990) for example has to predefine a set of 16 rules for synthezising a blocks-world problem. While we have only to transform an already given plan, Koza(1992) has to enumerate all possible programs which can be composed from a given set of primitive operators to synthesize the *tower* program.

Acknowledgments

We want to thank Eugene Fink, Jana Koehler, and Manuela Veloso for helpful discussions and Eugen Fink, Peter Geibel, Martin Mühlpfordt, and Heike Pisch for critical comments on an earlier draft of this paper.

References

Borrajo, D., and Veloso, M. 1996. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal* 10:1–34.

Christofides, N. 1975. *Graph Theory - An Algorithmic Approach.* London: Academic Press.

Cohen, W. 1998. Hardness results for learning first-order representations and programming by demonstration. *Machine Learning* 30:57–97.

Courcelle, B., and Nivat, M. 1978. The algebraic semantics of recursive program schemes. In Winkowski, ed., *Math. Foundations of Computer Science*, Springer LNCS. 16–30.

Dershowitz, N., and Jouanaud, J.-P. 1990. Rewrite systems. In Leeuwen, J., ed., *Handbook of Theoretical Computer Science*, volume B. Elsevier.

Field, A., and Harrison, P. 1988. *Functional Progamming.* Reading, MA: Addison-Wesley.

Flener, P., and Yilmaz, S. 1999. Inductive synthesis of recursive logic programs: Achievements and prospects. *Journal of Logic Programming* 41(2–3):141–195.

Kalmar, Z., and Szepesvari, C. 1999. An evaluation criterion for macro learning and some results. TR99-01, Mindmaker Ltd.

Kaplan, C., and Simon, H. 1990. In search of insight. *Cognitive Psychology* 22:374–419.

Klahr, D. 1978. Goal formation, planning, and learning by preschool problem solvers or: "my socks are in the dryer". In *Children's thinking: What develops?* Erlbaum.

Koehler, J. 1998. Planning under resource constraints. In Prade, H., ed., *13th ECAI.* Wiley.

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

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

Le Blanc, G. 1994. BMWk revisited: Generalization and formalization of an algorithm for detecting recursive relations in term sequences. In Bergadano, F., and de Raedt, L., eds., *ECML-94*, 183–197. Springer.

Manna, Z., and Waldinger, R. 1975. Knowledge and reasoning in program synthesis. *Artificial Intelligence* 6:175–208.

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

Minton, S. 1985. Selectively generalizing plans for problem-solving. In *Proc. of the IJCAI-85*, 596–599. Morgan Kaufmann.

Peot, M., and Smith, D. 1992. Conditional nonlinear planning. In *AIPS-92*, 189–197. Morgan Kaufmann.

Plotkin, G. D. 1970. A note on inductive generalization. *Machine Intelligence, Edinburgh Univ. Press* 5:153–163.

Precup, D., and Sutton, R. 1998. Multi-time models for temporally abstract planning. In *Advances in Neural Information Processing Systems*, vol. 10. MIT Press.

Schmid, U., and Wysotzki, F. 1998. Induction of recursive program schemes. In *Proc. of the ECML-98*, number 1398 in LNAI, 214–225. Springer.

Schmid, U. 1999. Iterative macro-operators revisited: Applying program synthesis to learning in planning. T. R. CMU-CS-99-114, Carnegie Mellon University.

Schoppers, M. 1987. Universal plans for reactive robots in unpredictable environments. In *IJCAI '87*, 1039–1046.

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

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

Smith, D. R. 1990. Kids: a semiautomatic program development system. *IEEE Transactions on Software Engineering* 16(9):1024–1043.

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

Sun, R., and Sessions, C. 1999. Self-segmentation of sequences: Automatic formation of hierarchies of sequential behaviors. T. R. 609-951-2781, NEC Research Institute, Princeton.

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

Veloso, M.; Carbonell, J.; Pérez, M. A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating planning and learning: The Prodigy architecture. *Journal of Experimental and Theoretical Artificial Intelligence* 7(1):81–120.

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

Wysotzki, F. 1987. Program synthesis by hierarchical planning. In Jorrand, P., and Sgurev, V., eds., *Artificial Intelligence: Methodology, Systems, Applications.* Elsevier. 3–11.