Iterative Macro-Operators Revisited: Applying Program Synthesis to Learning in Planning

Ute Schmid

March 1999
CMU-CS-99-114

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This paper was written while the author was on leave from the Department of Computer Science, Technical University Berlin, Germany at the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, invited by Jaime Carbonell and supported by a DFG research scholarship. The idea of combining planning and program synthesis was suggested by Fritz Wysotzki. Part of the work reported here was already done or prepared during the last year – supported by discussions with Fritz Wysotzki and by some work performed in studentical projects. I learned very much from Manuela Veloso when participating at her graduate course on planning in the spring semester 1999. I want to thank Eugene Fink, Jana Koehler, Avrim Blum and Jaime Carbonell for helpful and interesting discussions.

Keywords: macro generation, planning, inductive program synthesis

Abstract

The goal of this paper is to demonstrate that a method for inductive program synthesis (as described in [SW98]) can be applied to the problem of learning cyclic (iterative/recursive) macro-operations from planning. Input in the program synthesis system is a so-called initial program which represents an ordered set of straight-forward transformations from input states to the desired output. In the context of planning, the input states correspond to initial states, the output state to the planning goal, and transformations are shortest operation sequences. Ordering of transformations can be achieved by calculating a minimal spanning tree for the problem graph with the state(s) fulfilling the goal as root. We have implemented a non-linear backward planner which generates such a complete partial order as a by-product of planning. Output of the program synthesis system is a recursive program scheme representing the generalization of a program limited to solving a finite problem of given size to a general solution strategy. Our synthesis method is embedded in the theory of the semantic of functional programs and in the theory of inductive inference (see [MS98]) and thereby provides a sound formal basis for macro-construction. The current implementation can generalize tail, linear and tree recursive structures and combinations of such structures with multiple (and possibly interdependent) recursive parameters.

Contents

Chapter 1

Introduction

Learning macro-operators is a topic of interest nearly since the beginning of planning research [FN71a, Min85, Kor85]. While macros do not lead in general to greater efficiency in planning [Min85, DC96], they are nevertheless a useful and interesting contribution to planning research. Firstly, if construction of macros is not performed blindly but augmented by information about occurrence frequencies of operator sequences and if retrieval of macros is organized efficiently, planning can be speed-up considerably when introducing linear macros [Min85]. A linear macro-operator generally represents an aggregation of the pre- and post-conditions over a sequence of primitive operators. Using a macro-operator instead of a set of primitive operators reduces the number of match-select-apply cycles executed during planning and thereby also the number of planning decisions which might result in backtracking. For iterative macros efficiency gains should be even higher, because they not only provide aggregated operators but additionally incorporate part of the control strategy for planning [SC89, BV96].

Secondly, learning in planning is per se a topic of interest for cognitive science and artificial intelligence. Work on linear macro-operators in planning has its parallel in work on learning by chunking [RN86] and knowledge compilation [And86] in cognitive science. While this work addresses the fact that human problem solvers improve their skills by experience (i.e. faster generation of solutions which less errors in accordance with the power law of practice [NR81]), it neglects a second important aspect of human learning: When solving a problem of given complexity (as the Tower of Hanoi with three discs) humans can extrapolate a general strategy for solving problems of the same type with arbitrary complexity (as Tower of Hanoi problems with an arbitrary number of discs; [Kla78]). In production system architectures with fixed interpreter strategy this kind of learning cannot be modelled [SC89, SWar]. Therefore, coming up with a technique for learning cyclic macro-operators from experience is a challenging problem for planning as well as cognitive science research.[1]

---

[1] Now macro learning becomes also of interest in reinforcement learning. See NIPS'98

The importance of learning iterative macro-operators was first addressed by Carbonell, Cheng and Shell [CC86, SC89, DC96] and is also acknowledged by Schmid in the context of machine learning [SW98] and cognitive science [SWar]. While the work of Carbonell and colleagues addresses efficiency gains in planning and gives ideas about a technique for constructing iterative macros in context of the expert system FERMI, Schmid proposes to combine planning and program synthesis in the following way: Planning provides straight-forward "programs" for transforming input into desired output states. These straight-forward programs can be generalized by a technique for inductive program synthesis. The first step corresponds to exploring a problem or – in the context of program construction – to hand-simulating I/O transformations. The second step corresponds to (unsupervised) learning from experience and – in the context of automatic programming – to programming by demonstration [Coh98, Wel99a]. While the first step is dependent on background-knowledge (especially on the semantics of predefined operations and on knowledge about data structures), the second step can be performed (with some limitations, see [MS98]) purely syntactically. That is, generation of initial programs is dependent on heuristic information – but the synthesis method corresponds to a generic pattern-matching algorithm which can be completely formalized and its soundness, efficiency and scope can be determined in a general way.

In this paper we will focus on generating initial programs from planning. The synthesis technique is described in detail elsewhere [SW98, MS98]. Because our approach is somewhat out of the context of current research in planning, we start with an informal example to motivate our ideas. More concrete, we will show that the function *clearblock* can be easily inferred from planning experience. The *clearblock*-function is an example of a simple iterative macro [MW87], corresponding to a linear recursion. Afterwards, we will introduce our planning system and discuss it in relation to other planning systems as PRODIGY [VCP$^+$95], UCPOP [PW92], graphplan-approaches [BF97, KNH97] and universal planning [Sch87, Sch95, CRT98]. Our system DPlan is a sound and complete non-linear backward planner implemented in LISP. In the fourth part, we will give further examples of learning linear-recursive macros, as for example a general solution strategy for the rocket-problem with an arbitrary number of objects [VC93b] and afterwards we will discuss more complex examples as constructing a tower of ordered blocks and Tower of Hanoi; furthermore we will address the application of planning to list and number problems. We will conclude with an evaluation of our approach and further work to be done.

---

workshop "Abstraction and Hierarchy in Reinforcement Learning": Amy McGovern, University of Massachusetts, Amherst, acQuire-macros: An Algorithm for Automatically Learning Macro-Actions, and David Andre, University of California, Berkeley, Learning Hierarchical Behaviors; http://www-anw.cs.umass.edu/~ dprecup/call_for_participation.html.

Chapter 2

How to clear a block – inductive generalization of plans

2.1 A recursive function for clearing a block

A human problem solver in many cases is able to generalize a strategy for solving a complete class of problems from his/her experience of solving some example problems. For example, if a person is able to clear the bottom block of a three block tower, we assume that he/she also is able to clear the bottom block of a tower consisting of an arbitrary number of blocks, even if the person has never performed this action sequence before. The need of automated mechanisms for this kind of learning by experience was for example addressed more than a decade ago by Manna and Waldinger [MW87]. Our proposal differs in two aspects from this seminal work: firstly, while Manna and Waldinger discuss the synthesis of imperative programs, we will discuss the synthesis of functional programs; secondly – and more crucially – Manna and Waldinger proposed a *deductive* technique for synthesizing conditional and recursive plans while we are proposing an *inductive* technique, based on inductive program synthesis.

The plan Manna and Waldinger can derive automatically from a goal specification by their deductive tableau method is

$$makeclear(a) \Leftarrow \begin{cases} \textbf{if} & clear(a) \\ \textbf{then} & \Lambda \\ \textbf{else} & makeclear(hat(a)); \\ & put(hat(a), table). \end{cases}$$

To clear a block a, first determine whether it is already clear. If not, clear the block that is on top of block a and then put that block on the table. If a block a is clear, the empty sequence of operations Λ is returned.

Our functional variant of this program is

$$clearblock(x, s) = g(cleartop(x), s, puttable(topof(x), clearblock(topof(x), s)).$$

We represent conditionals by the McCarthy-conditional $g(x, y, z) \equiv$ *if x then y else z*. To clear a block x in a situation s, check whether x is already clear. If yes, do nothing and return the current situation; otherwise, put the block lying on top of block x on the table in a situation where this block is already cleared. The situation variable s holds a conjunction of propositions, as {*on(A B), on(B C), clearblock(A)*} where A, B and C are constants. The predicate *cleartop(x)* is true, if x is currently instantiated with a constant K and *clearblock(K)* is an element of the current situation s. The function *topof(y)* is defined as *on(x y)* $\equiv$ *topof(y) = x)*. Application of the operator *puttable(y,s)* has the effect that *on(y z)* is deleted and *clearblock(z)* is added from/to the propositions representing situation s. The function *clearblock(x,s)* is linear recursive, that is, it corresponds to a loop or – in other words – to an iterative macro-operation. In contrast to the iterative macros proposed by Shell and Carbonell [SC89], in our case, if the precondition is fulfilled the macro terminates, otherwise, the operation *puttable* is (repeatedly) applied.

Excursion: Inductive synthesis of recursive program schemes

We will present the general idea of our synthesis method in a nutshell. The method was originally proposed by Wysotzki [Wys83] as an extension of Summers' method [Sum77] and was extended further, implemented and formalized in the framework of grammar inference by Schmid [SW98, MS98].

The synthesis technique starts with an initial program as input and tries to fold it by extrapolating a recursive program scheme. An initial program is a nested conditional expression representing an ordering of straight-forward transformations. We can synthesize the *clearblock*-function from the following initial program:

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

This initial program represents the experience of clearing a block in a world consisting of maximally three blocks. If there are more blocks (i.e. $topof(topof(x))$ is not clear), the actions are undefined (Ω). Note, that the initial program corresponds to the second unfolding of the *clearblock*-function given above. That is, for program synthesis we reverse the idea of determining the semantic of a recursive function as its smallest fixpoint [FH88]: from a given sequence of unfoldings we want to extrapolate the minimal recursive program scheme which can generate these unfoldings.

An initial program G can be folded into a recursive program iff it can be decomposed into a sequence $\mathcal{G}^{(0)} = \Omega$, $\mathcal{G}^{(l)} = tr(\mathcal{G}^{(l-1)}_{[t/v]}/m)$ with $l = 1 \ldots n$, $\mathcal{G}^{(n)} = G$ of partial transformations which successively cover a larger amount of inputs. That is, we have to find a term tr in G which has the characteristic that successively more complex subterms of G can be

> expressed by inserting the previous subterm at position m where variables v are substituted by terms t. For our example we have:
>
> $$\begin{aligned} \mathcal{G}^{(0)} &= \Omega \\ \mathcal{G}^{(1)} &= g(cleartop(x), s, puttable(topof(x), \Omega)) \\ \mathcal{G}^{(2)} &= g(cleartop(x), s, puttable(topof(x), \\ &\quad g(cleartop(topof(x)), s, puttable(topof(topof(x)), \Omega)))) \\ &= G \end{aligned}$$
>
> with *tr = g(cleartop(x), s, puttable(topof(x), m))* and the substitutions $[topof(x)\ /x]$.
>
> Because $\mathcal{G}^{(l)} = tr(\mathcal{G}^{(l-1)}_{[t/v]}/m)$ holds for all $\mathcal{G}$'s,
>
> $$\begin{aligned} \mathcal{G}^{(0)} &= \Omega \\ \mathcal{G}^{(1)} &= g(cleartop(x), s, puttable(topof(x), \mathcal{G}^{(0)}_{[topof(x)/x]})) \\ \mathcal{G}^{(2)} &= g(cleartop(x), s, puttable(topof(x), \mathcal{G}^{(1)}_{[topof(x)/x]})) \end{aligned}$$
>
> we can fold G into $\mathcal{G} = tr(\mathcal{G}_{[t/v]}/m)$. For our example that is
>
> $$\mathcal{G} = g(cleartop(x), s, puttable(topof(x), \mathcal{G}_{[topof(x)/x]}))$$
>
> which corresponds to the *clearblock*-program given above.
>
> Our method is defined independently of a given programming language. Instead, we consider programs as terms which are elements of some arbitrary term algebra and we infer recursive program *schemes* [CN78] which can be interpreted with respect to a programming language. With our method we can infer tail recursive structures (i.e. for-loops), linear recursive structures (i.e. while-loops), tree-recursive structures and combinations thereof. We can deal with programs starting with a constant part and – what is beyond the scope of ILP-approaches [MDR94, FYar] – we can deal with multiple recursive parameters which can be interdependent. What is currently out of the scope of our method are substitutions which are themself performed recursively (as in the *ackerman*-function). For details about the formal background, the synthesis algorithm, its scope and complexity, see [SW98, MS98].

Our method of program synthesis from initial programs can be seen as an approach to learning by generalizing over experience or – in the context of programming – as programming by demonstration. From the perspective of planning research, we argue that planning provides initial experience with a problem (i.e. a kind of initial program) which we can generalize by means of program synthesis. That is, we propose a method which makes it possible to infer a large class of cyclic macros in a formally sound way. From the perspective of program synthesis research, we argue that the method to construct initial programs by rewriting input/output examples (see below) can be replaced by more powerful (and more natural) planning techniques. To sum up, we propose the following steps for learning cyclic macro-operators:

1. Constructing initial programs from input/output examples:
    (a) Construct an ordered set of straight-forward transformations from input states to output (i.e. goal) states by planning
    (b) Transform the plan into an initial program
2. Generalizing initial programs into cyclic macro-operations by program synthesis.

Above, we showed, how the cyclic macro for the *clearblock*-problem can be inferred from an initial program. In the rest of this section, we will describe the first step – constructing an initial program by planning – for the *clearblock*-problem.

2.2 A plan for clearing a block in a finite problem space

In the following we will show how the initial program for the *clearblock*-problem can be automatically constructed by planning. For the simple *clearblock*-problem we need only one operator – *puttable* – which we define in a STRIPS-like syntax:

```
(setq rules '( (rule puttable
                 (if ( ct (> x) ) ( on (< x) (> y) ) )
                 (then (puttable (< x)))
                 (conq ( add ((ct (< y)))
                         del ((on (< x) (< y))) ) )
) )).
```

The presented notation is the one used for our planner DPlan: The preconditions are given as a list starting with the keyword *if* (with *ct* as shorthand notation for *cleartop*). The preconditions are interpreted as conjunction of positive literals with existential quantified variables. Variables are represented by $(> x)$ signaling that the binding for x is to be obtained from the current state or $(< x)$ signaling that x has to be instantiated in accordance with the current bindings. The operator is given after the keyword *then*. The operator effect is given as usual by add- and del-lists (which represent conjunctions of positive literals with existential quantified variables). Our planner can also deal with conditional effects which we will show in the next section.

For our example we assume a blocksworld consisting of three blocks A, B and C. Our planning goal is

```
(setq goal '( (ct C) ) ).
```

DPlan can deal with goals which are composed of a conjunction of (possibly interdependent) subgoals which will also be discussed in the next section.

To be able to generalize, we need to make the following experience: clearing C when it is the bottom of a three-block tower, clearing C when it is the bottom of a two-block tower and clearing C when it is already clear. That is, we want to regard for example the following initial states:

```
(setq states '(
   ((on A B) (on B C) (ct A))
   ((on B C) (ct A) (ct B))
   ((ct A) (ct B) (ct C))
)).
```

Typically, a planner would produce a separate plan for each of the initial states. In contrast, we want to deal with all three states in a single planning process and thereby obtain a plan covering all given initial states[1]. The idea, to use planning over sets of states in program synthesis was originally proposed by Wysotzki [Wys87].

Informally, our planner proceeds in the following way:

1. Search for all states in `states` which are subsumed by `goal` (i.e. where the set of goal literals is a subset of a state).
2. If there are no such states: stop with error (the goal cannot be fulfilled) else delete these states from `states` and use them as root for the plan (If there is only one state fulfilling the goal, use this state as root, otherwise introduce a top-node with all states fulfilling the goal as children.).
3. For each node in the plan do recursively until `states` is empty:
    (a) Calculate all immediate predecessors (by backward-application of operator effects). Insert the operations as arc-labels and the predecessors as new nodes in the plan[2].
    (b) Delete all states corresponding to the predecessors from `states`.

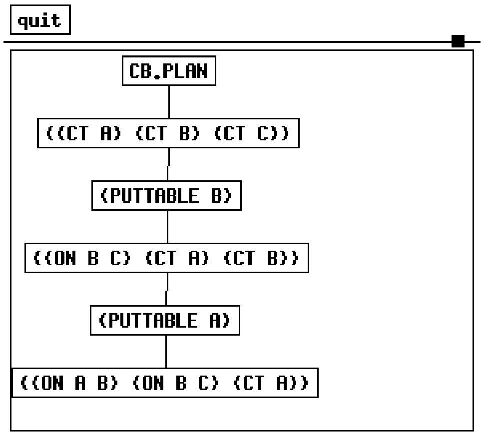
Deleting all states corresponding to calculated predecessors from the set of initial states guarantees that only the shortest possible (optimal) operation sequences to transform a state into the goal are calculated and that there occur no cycles. Furthermore, the planner can work completely without backtracking. The resulting plan is a complete partial order over the initial states, i.e. the minimal spanning tree [GH85] of the problem space with the goal state(s) as root. Each planning step is saved as a structure of instantiated operation, predecessor node, constructed new state (child node), and the instantiated preconditions and effects.

To generate (shortest) operation sequences for transforming a set of n initial states into the desired goal, in classical approaches planning has to be performed n times. Each planning episode could involve backtracking. We are arguing, that regarding a set of states "simultaneously" in one backtracking-free planning process is more efficient than planning n times with backtracking. Furthermore,

---

[1] We will discuss the relation of DPlan to universal planning in the next chapter.

[2] This step will be discussed in detail in the next chapter.



```
CB.PLAN
((CT A) (CT B) (CT C))
(PUTTABLE B)
((ON B C) (CT A) (CT B))
(PUTTABLE A)
((ON A B) (ON B C) (CT A))
```

Figure 2.1: Output of DPlan for *clearblock*

by providing the planner with a set of states, there is no need of providing axioms for checking the admissibility of predecessor states. Instead, we can determine if a new state is legal by a look-up in the set of states. Finally, if generating predecessor states involves instantiation of free variables (as y in puttable) the set of legal instantiations can also be calculated by look-ups in the set of states. In section 3 we will present these aspects of DPlan in detail.

The output DPlan generates for the *clearblock*-problem is given in figure 2.1[3]. Only one state fulfilled the goal `(ct C)`. This state is the root of the plan. Only one operation – `(puttable A)` – is applicable. Its backward application results in a single legal predecessor state. Again, only one operation – `(puttable B)` – is applicable. The next predecessor cannot be expanded further. All states from `states` have been used in planning. The plan tree (only a list for this simple example) represents the shortest operator sequences for transforming each initial state into the goal: `((CT A) (CT B) (CT C))` fulfills the goal already, the other states can be transformed by: *puttable(A, ((on B C) (CT B) (CT A)))* and *puttable(B, puttable(A, ((on A B) (on B C) (CT A))))*.

2.3 Replacing constants by constructive expressions

In the rest of this section we will describe how the plan produced by DPlan is transformed into an initial program. The crucial aspect of this transformation is to replace constants by constructive expressions. To motivate this step, we

---

[3] All graphics are the original outputs produced by DPlan. Operations are regarded as arc-labels but are represented as intermediate nodes between two states in the graphical output of DPlan.

make an excursion to program synthesis again.

An excursion to Summers' method

In Summers' [Sum77] classical approach to inductive program synthesis, the first step is to rewrite the given input/output examples constructively. To use Summers' own example (and notation), if we have the I/O-pairs

$$\begin{array}{rcl} (\ ) & \rightarrow & (\ ) \\ (A) & \rightarrow & ((A)) \\ (AB) & \rightarrow & ((A)(B)) \\ (ABC) & \rightarrow & ((A)(B)(C)) \end{array}$$

the righthand-sides (outputs) are rewritten into functional expressions for transforming the input into the output

$$\begin{array}{rcl} \{f_1[x] & = & nil; \\ f_2[x] & = & cons[x; nil]; \\ f_3[x] & = & cons[cons[car[x]; nil]; cons[cdr[x]; nil]]; \\ f_4[x] & = & cons[cons[car[x]; nil]; cons[cons[cadr[x]; nil]; \\ & & \quad cons[cddr[x]; nil]]]\}. \end{array}$$

Rewriting is performed by applying the predefined functions *car*, *cdr* their legal compositions and *cons* to the input-part of the example in such a way that they transform the input into the desired output[4]. The I/O-examples can now be written as

$$\begin{array}{rcl} \{() & = & nil; \\ (A) & = & cons[x; nil]; \\ (AB) & = & cons[cons[car[x]; nil]cons[cdr[x]; nil]]; \\ (ABC) & = & cons[cons[car[x]; nil]cons[cons[cadr[x]; nil] \\ & & \quad cons[cddr[x]; nil]]]\}. \end{array}$$

This corresponds to the minimal spanning tree generated by DPlan: for each state (input example) we are provided with the shortest operation sequence to transform the input into the goal (desired output) and these sequences are ordered by their complexity. We will show later, that planning provides a far more powerful method for rewriting I/O-examples than the technique proposed by Summers.

After rewriting the output-parts of the examples, we need a more abstract way to discriminate between the different inputs. The equations above express statements as "if parameter x of function $f(x)$ has value '(A)' then return (*cons x nil*)". Now we want to replace the input values by boolean expressions as "if x is a one-element list". To obtain this,

---

[4] Summers' method was restricted to functions with a single list as argument which return a list by using only the primitive operations *car*, *cdr* and *cons*. Rewriting was restricted much more than it is the case in genetic programming (see for example [Koz92] for the synthesis of the "tower" program).

Summers relies on a predefined complete partial order (cpo) over lists (with the empty list as bottom-element). He abstracts from the concrete inputs and classifies their abstract forms with respect to the cpo. For example, the form of $(A\ B)$ is $(\omega\ \omega)$ with $(\omega) < (\omega\ \omega) < (\omega\ \omega\ \omega)$ can be characterized by $atom[cddr[x]]$. We omit the details given in [Sum77]; for our example we obtain:

$$\begin{aligned} p_1[x] &= atom[x] \\ p_2[x] &= atom[cdr[x]] \\ p_3[x] &= atom[cddr[x]] \\ p_4[x] &= T. \end{aligned}$$

The first step of Summers' program synthesis technique is finished when the $p_i \to f_i$ expressions are combined into a single function

$$\begin{aligned} F[x] \quad \leftarrow \quad & [atom[x] \to nil \\ & atom[cdr[x]] \to cons[x; nil] \\ & atom[cddr[x]] \to cons[cons[car[x]; nil]; cons[cdr[x]; nil]] \\ & T \to cons[cons[car[x]; nil]; cons[cons[cadr[x]; nil]; \\ & \quad cons[cddr[x]; nil]]]]. \end{aligned}$$

This is the kind of initial program, Summers constructs from I/O-examples. In the next step, the initial program is folded into a recursive program in a similar but more restricted way than our method (see section 2.1). By detecting regularities between succeeding subexpressions, the following program for "unpacking" lists is synthesized:

$$\begin{aligned} unpack[x] \quad &\leftarrow \quad [atom[x] \to nil; \\ & \qquad\ T \to u[x]] \\ u[x] \quad &\leftarrow \quad [atom[cdr[x]] \to cons[x; nil]; \\ & \qquad\ T \to cons[cons[car[x]; nil]; u[cdr[x]]]]. \end{aligned}$$

To come back to the *clearblock*-problem: Planning provided us with (nearly) the righthand-side of such straight-forward programs. For our *clearblock* example (cf. figure 2.1) we have:

$$\begin{aligned} f_1[x, s] &\to s \\ f_2[x, s] &\to puttable(B, s) \\ f_3[x, s] &\to puttable(A, puttable(B, s)). \end{aligned}$$

In contrast to Summers, we still have the constants given in the states instead of more abstract expressions. Summers used knowledge about the structure of lists and a predefined order over their complexity to rewrite inputs as *(A B)* into *atom[cddr[x]]*. We will do something similar to this idea: providing our system with background knowledge about the structure of the problem domain in the form of rewrite-rules. For the *clearblock*-problem we provide the following information

```
; Predicates which indicate constructive rewriting
(defun is-c-pred (p-name)
  (equal p-name 'on)
)

; rewrite rule
; p is a proposition (predicate-name arguments)
; c-eq returns an association list
; of constant - constructive-expression pairs
; (on x y) == [x = (topof y)]
(defun c-eq (p)
  (list (nth 1 p) (list 'topof (nth 2 p))) ; assoc-list
).
```

A proposition $on(K_1, K_2)$ gives us the information that constant K_1 is on the top of constant K_2. So, we use all *on*-predicates in a linear (sub-) plan to generate bindings of constants with their constructive equivalent. This idea corresponds to Manna and Waldinger's use of the *hat*-axiom [MW87]:

> *if not* $Clear(w, y)$
> *then* $On(w, hat(w, y), y)$ for all states w and blocks y.

For the *clearblock*-problem we obtain the bindings *((A = topof(topof(C))) (B = topof(C)))* by applying the following (here only informally given) algorithm

- Find substitutions
  - For each proposition p of each state Do
    - If `is-c-pred (predicate-name of` p`)` then return `(c-eq p)`(i.e. the binding of the constant(s) with their constructive equivalent)
    - Union all found bindings
- Apply substitutions
  - For each proposition of each state Do
    - As long as applying the substitutions results in a different form of the arguments of the proposition Do
      replace the arguments by their constructive equivalence.

"Find substitutions" gives us *((A = topof(B)) (B = topof(C)))*; and "Apply substitutions" rewrites all propositions of each state recursively. That is, $topof(B)$ gets rewritten to $topof(topof(C))$. For problems more complex than *clearblock* rewriting is done for each linear subplan (see chapter 3). The output generated by DPlan is given in figure 2.2. Now – as for the righthand sides of Summers' straight-forward programs – the operations are represented in a constructive way and – what we will need for constructing the boolean conditions – the states itself are represented over a constructive data type also. The remaining constant symbols in the plan can now be interpreted as variables (in correspondence to Summers' abstract forms of the original inputs).

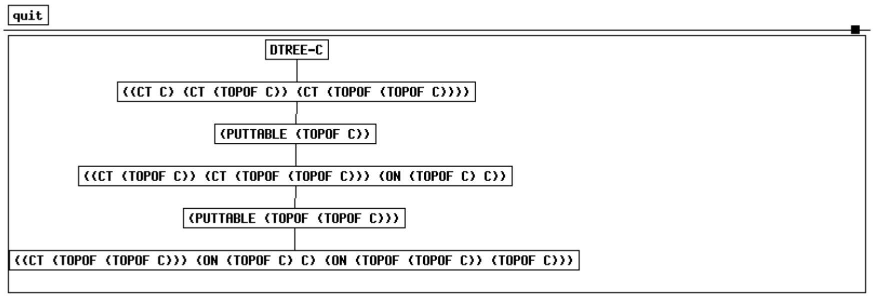


Figure 2.2: Result of constructively rewriting constants for *clearblock*

Using background-knowledge – as Summers' cpo or Manna and Waldinger's axioms – is an important technique for learning cyclic macro-operators in planning. While background knowledge is seldomly regarded in classification learning [Mit97] it is central (domain theory) in explanation based learning [MKKC86] and also common practice in the context of program synthesis from examples (see cf. the inductive logic programming literature [MDR94, FYar] and the genetic programming literature [Koz94])[5].

2.4 Determining relevant predicates

In the next step, will determine the minimal set of relevant predicates of each constructively rewritten state of the plan. This corresponds to Summers' introduction of boolean expressions as the lefthand-sides of his straight-forward programs – and ins some way also to the discrimination predicates used in conditional planning [PS92, BV97]. The relevant predicates can be identified easily from the information obtained during planning: Each legal planning step consists of an instantiated operator which transforms a state (the state newly constructed by backward-planning) into the state given at the current node. That is, the predicate(s) in the add-list of the operator must match with predicate(s) of the current state. These predicates are the one which are fulfilled in the current planning step and therefore, they are the relevant predicates.

For the *clearblock*-example we obtain: *(ct C)* for the root-node which is generated by *puttable(topof C)* and *(ct (topof C))* for the second node which is generated by *puttable(topof(topof C))*. The leafs of a plan represent states for which exists no predecessor in the given problem domain. For leaf nodes, we regard the predicates given as preconditions of the last applied operator. Thus, we obtain {*(ct (topof(topof C))), (on (topof(topof(C)) (topof C)))*} as relevant

---

[5] In some planning systems, cf. PRODIGY [VCP+95], the planner can be provided with additional information about control rules to make planning more efficient for a given domain; similarly, such systems could be extended to provide information about the data structures of the planning domain

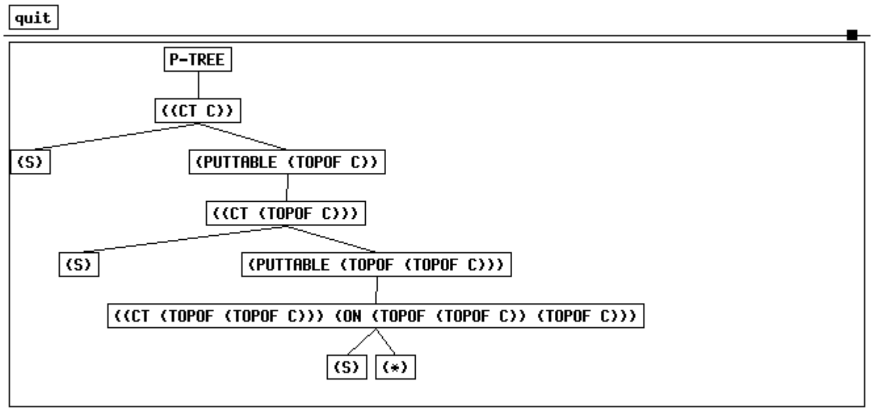


Figure 2.3: Binary planning tree for *clearblock*

predicates for the last state. But the predicate (*on (topof(topof(C)) (topof C))*) is not really relevant. It plays no role with regard to a general strategy for clearing a block. There are at least two heuristic strategies for cleaning the leaf-node predicates: First, we could assume that, if up to now only the operator *puttable* was used in planning this would also be the case for larger domains and accordingly replace the leaf-predicates by the instantiated add-list of *puttable*. Secondly, we could compare the predicates used so far and only accept such predicates of the preconditions which are also used in the predecessor nodes. But we will see, that we don't need such heuristics because the leaf states are not regarded when rewriting the plan to an initial program.

2.5 Generating a binary tree

Now we are nearly done with transforming the original plan tree into a format suitable for generalization. We will rewrite the plan into a binary structure. This structure is similar to the plans proposed by [Wys87] and can be used to generate the shortest operator sequences for a set of input states (similar to a universal plan) by means of an interpreter function.

In case of linear plans, generating the binary tree is simple: For each predicate node we introduce a left successor s. The variable s represents that there is a situation where the predicate given at the predecessor node is true, or, more generally, that the predicates given on the path to this leaf are true. The right successor is the next predicate node with the arc labelled by the operation. If there is no further operation given, we terminate the path with an asterix, indicating, that for this case, we have made no experience during planning. The binary tree for the *clearblock*-problem is given in figure 2.3.

In case of non-linear plans, we try to unify different paths. If there still remain non-linear subplans, we can either introduce an arbitrary sequence or we can try to generalize over the tree structure (see chapter 4). Note, that for unifying different paths of a plan it is crucial that constructive rewriting has been performed *before*: when searching a generalization, it is meaningless to unify sub-paths involving identical constants. But it makes sense to unify sub-paths with identical constructive expressions!

2.6 Transforming the plan tree into a program term

In the last step, we rewrite the final plan into the syntax required for initial programs as input in our program synthesis system[6]. To obtain the initial program for *clearblock* given above

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

we regard each predicate node as a boolean condition, its left successor as the then-case and its right successor as else-case. Remember that the remaining constants in the rewritten plan are regarded as variables.

The plan can be read as "If block C is clear then return the current situation, else, put the top of block C on the table in a situation for which holds: if the top of block C is clear then ...". So, if an arc is labelled with an operator, the subtree becomes the additional argument of the operator. If the subtree of an operator does not contain a further operator and if there is an asterix in this subtree, we terminate rewriting by introducing Ω (representing the undefined sequence in infinite term algebras [Wys83, CN78]) as argument.

Note, that the *clearblock* macro not only generalizes over the size of the blocksworld domain (number of blocks), but also over the position of the block which is to be cleared. In planning, we made experience with clearing the bottom block of a tower, the recursive program can clear an arbitrary block x.

---

[6] Remark: this step is not implemented yet. Currently, we provide "handmade" initial programs or unfoldings of given recursive programs to our synthesis system. Eckhard Wiederhold now is implementing the transformation as part of a student project.

Chapter 3

DPlan – A non-linear backward planning system

It may be a little unusual to propose a non-linear backward planning algorithm when the state of the art is graphplan [BF97, KNH97] and SAT-planning [KS98]. But from the perspective of *learning in planning* efficient *plan construction* is not the crucial question. From a cognitive point of view [SWar], we want to model how experience with a small finite problem domain can be generalized to an efficient solution strategy for a complete class of problems. It is not very plausible to assume that a human problem solver would explore a logistics problem with 105 action steps [Wel99b]. In fact, we can assume that it is possible to extract a general strategy as "try to load as many objects as possible in the transporter if at one place" from very small problem domains (see transportation example in section 4). From the perspective of planning we propose an automated approach to generalize finite plans into cyclic macro-operations. For our synthesis method described shortly in section 2.1, initial programs which correspond to the third expansion of a hypothetical recursive program scheme are sufficient [MS98]. So, usually it is enough to construct plans in domains consisting of three objects of each given type and clearly for domains of such restricted size efficiency questions are not crucial. On the other hand, we argue that the availability of iterative macro-operations can lead to high efficiency gains when the planner is confronted with a problem of a already known class (cf. Tower of Hanoi with 3 discs or sorting of lists with three elements) but of different size (n discs, n list elements). Similar arguments as well as calculations of efficiency gains and experimental data for iterative macro-operators are presented by Shell and Carbonell [SC89].

In this chapter we will present the planning system DPlan and discuss it in relation to current planning methodology. The transformation of the originally output of DPlan into an initial program is discussed in chapter 4.

3.1 Semantics for D-plans

The planner DPlan produces the minimal spanning tree of a finite problem space (for an overview of algorithms for calculating spanning trees of graphs see cf. [GH85, Tar83]). The root of the spanning tree are all problem states which are subsumed by the planning goal. The minimal spanning tree represents the shortest operation sequences for all problem states. Because this idea is similar to the Dijkstra-algorithm for calculating all shortest paths to a given node in a graph [Dij59], we call our planner DPlan and the resulting plans D-plans.

3.1.1 Operators, states, and operations

First, we define operators, states, operations (actions)[1].

Definition 1 (Operator). An operator is a 4-tuple consisting of

1. a **name**, which is a string,
2. the **precondition** φ_0, which is a conjunction of positive literals,
3. an **operation template** $\hat{o}$, which is the operator name and the number and names of its arguments,
4. the **effect** $\varphi \Rightarrow \alpha_t\delta_t, \alpha_e\delta_e$, with φ as effect condition (limited to a single positive literal), α_t and δ_t as Add- rsp. Del-effects if φ is true and α_e and δ_e as Add- rsp. Del-effects if φ is false.
   When $\varphi = \emptyset$, $\alpha_t = \emptyset$, $\delta_t = \emptyset$, the operator is "unconditioned".

Note, that in contrast to the definition of operators in other planning systems [PW92, VCP+95, KNH97] we do not give an explicit parameter list. Up to now, we have only untyped parameters. Furthermore, we currently allow only a single effect condition. In the current (first) implementation of the planner we do not regard negation, disjunction and universal quantifiers. We hope to extend our approach to these features.

An example of the valid operator *put* is given in figure 3.1. This operator has a conditional effect: A block x can be put on a block y if both blocks are clear. Applying this operation has the (primary) effects [FY95] that x lies on y and y is no longer free. If block x was on the table, these are the only effects; but, if x was lying on another block z, there are the additional (side-) effects that x does no longer lay on z and z is clear. Figure 4 shows only *one* possibility for representing the *put*-operator. Another possibility is, to introduce an additional predicate $ontable(x)$.

Definition 2 (State). A **state** is a set of positive ground literals (ground atoms). We denote a positive ground literal with p and the set of positive ground literals with P. As usual, a state $S \in 2^P$ is a set of ground atoms.

Examples for states are $\{on(A,B), on(B,C), ct(A)\}$, $\{ct(A), ct(B), ct(C)\}$.

---

[1] Terminology and sequence of our definitions follow mostly [KNH97].

name: put

pre : {cleartop(x), cleartop(y)}

op : put(x,y)

eff : if on(x,z)
then ADD {ct(z), on(x,y)} DEL {on(x,z), ct(y)}
else ADD {on(x,y)} DEL {ct(y)}

Figure 3.1: The operator *put*

name: put

pre : {cleartop(x), cleartop(y), for-all(z) ¬ on(x,z)}

op : put(x,y)

eff : ADD {on(x,y)} DEL {ct(y)}

name: put

pre : {cleartop(x), cleartop(y), on(x,z)}

op : put(x,y)

eff : ADD {ct(z), on(x,y)} DEL {on(x,z), ct(y)}

Figure 3.2: The two variants of the *put*-operator

Definition 3 (Operation/Variants of an operator). An **operation** o is a ground instance of an operator. That is, all parameters are instantiated with constants. Additionally, operations have unique effects. A conditioned operator (as defined in def. 1) results in two operations: $\varphi_0 \cup \varphi; \alpha_t, \delta_t$ and $\varphi_0 \cup \neg\varphi; \alpha_e, \delta_e$. We call the uninstantiated forms of a conditioned operator **variants**.

We use the term operation instead of the often used term *action*. In contrast to other planners dealing with operators with conditioned effects, in DPlan each operation has a unique effect. Instantiation of a conditioned operator is performed with respect to its variants. The variants of the *put*-operator are given in figure 3.2. Note, that while DPlan does currently not allow for negation and universal quantifiers in specifying *operators* we can deal with such expressions in operator *variants*[2]. To apply α_e, δ_e we have to make sure that $\neg\varphi$ holds *for all* possible instantiations.

An example of two *put*-operations derived by instantiating the *put*-operator

---

[2] Of course, we have not really to deal with universal quantifiers because $\forall x \neg p(x) \equiv \neg\exists p(x)$.

o_1 is valid for $\{ct(A), ct(B), ct(C)\}$ in a blocksworld of three blocks A, B, C

name: put

pre : {cleartop(B), cleartop(C), ¬ on(B,A)}

op : put(B,C)

eff : ADD {on(B,C)} DEL {ct(C)}

o_2 is valid for $\{on(B, A), ct(B), ct(C)\}$ in a blocksworld of three blocks A, B, C

name: put

pre : {cleartop(B), cleartop(C), on(B,A)}

op : put(B,C)

eff : ADD {ct(A), on(B,C)} DEL {on(B,A), ct(C)}

Figure 3.3: Operations derived from the *put*-operator

is given in figure 3.3. In a blocksworld consisting of only three blocks A, B, and C, the preconditions φ_0 constrain the possible instantiations of z. For the examples in figure 3.3 we have $\sigma = \{z \leftarrow A\}$.

3.1.2 Operation applications, admissible states, and variable bindings

In the following we will define the application of a single operation and of operation sequences. Furthermore, we will describe how the admissibility of calculated predecessor states is checked, when operations are applied backward and we will describe how variable bindings are determined in backward-planning. Remember, that our planner works for *sets of initial states*, i.e. it calculates a cpo for all states of a finite problem domain.

Definition 4 (Applying an operation). We denote the set of all operators with $\mathcal{O}$. Let O be the set of all ground instances (operations) of $\mathcal{O}$ and $Res : 2^P \times O \rightarrow 2^P$ be a function from states and operations to states.
Application of a single operation o to a state S is defined as

$$Res(S, o) = \begin{cases} (S \cup A(S, o)) \setminus D(S, o) & ;\ if\ \varphi_o \subseteq S \\ undefined & ;\ otherwise \end{cases}$$

with $A(S, o)$ as instantiated ADD-list of o and $D(S, o)$ as instantiated DEL-list.

Definition 5 (Applying an operation-sequence). The function Res can be expanded to $\overline{Res} : 2^P \times O^* \rightarrow 2^P$ with O^* as set of operation-sequences. Application of a sequence of operations can be defined recursively as

$$\overline{Res}(S, < o_1, \ldots o_n >) = \begin{cases} Res(S, o) & ; if \ \| < o_1 \ldots o_n > \| = 1 \\ Res(\overline{Res}(S, < o_1, \ldots o_{n-1} >), o_n); & otherwise. \end{cases}$$

The forward application of operation-sequences is used in *plan execution.*

Because our planner is working *backward*, starting from the goal, we also have to define backward application of an operation and backward application of a set of operations.

Definition 6 (Applying an operation backward). We denote the set of all operators with $\mathcal{O}$. Let O be the set of all ground instances (operations) of $\mathcal{O}$ and $Res : 2^P \times O \to 2^P$ be a function from states and operations to states. Application of a single operation o **backward** to a state S is defined as

$$Res^{-1}(S, o) = \begin{cases} (S \cup (D(S, o) \cup \varphi_0)) \setminus A(S, o) & ; if \ A(S, o) \subseteq S \\ undefined & ; otherwise \end{cases}$$

with $A(S, o)$ as instantiated ADD-list of o and $D(S, o)$ as instantiated DEL-list.

In contrast to forward-planning, we have to check whether a predecessor state S' calculated by $Res^{-1}(S, o)$ is *admissible*, i.e. if it is a legal state in the planning domain. In our context, a state is admissible if it is consistent. For example, a state in which block A lies on block B and block B is *cleartop* is not admissible because it contains contradictory propositions.

Admissibility can be checked via axioms (see [Wys87] and model-based planning [CRT98]). That is, $S = \{p_1 \ldots p_n\} \in 2^P$ is admissible if there is no axiom where from some p_i's $\in S$ follows $\neg p_j$ and $p_j \in S$. Another possibility to check admissibility is, to generate a candidate predecessor S' by backward application of operation o and check whether $Res(S', o) = S$ for the current state S.

We can exploit the fact that our planner works on sets of initial states (i.e. all states of a given finite problem domain). Because all states of a problem space and no other states are admissible for a giving planning problem we can omit the introduction of axioms and defined instead

Definition 7 (Admissibility of states). For a given planning problem in problem space $\mathcal{D}$ a state S is **admissible** iff $S \in \mathcal{D}$.

We illustrate the calculation of admissible predecessor states by backward application of an operation, again with the familiar blocksworld example (see figure 3.4). For the clarity of the example we assume that only the *put*-operator given in figure 3.1 is available (for a complete specification of the *tower*-problem we have to provide additionally a *puttable*-operator, see section 2.2 and below). We obtain four instantiated operators consistent with the current state S. For the three-block world, variable z can only be instantiated with C for put_1 and put_2 and only with A for put_3 and put_4. Only the application of put_1 leads to an admissible predecessor state. For put_2 subtraction of the Add-list from S can not be performed completely, because $ct(C) \notin S$ and there is no state in $\mathcal{D}$

Set of all admissible states of the three-block blocksworld:
$\mathcal{D}_{tower}$ = { {on(A B), on(B C), ct(A)}, {on(B C), ct(A), ct(B)}, {ct(A), ct(B), ct(C)}, {on(B A), on(A C), ct(B)}, {on(C A), on(A B), ct(C)}, {on(A C), on(C B), ct(A)}, {on(B C), on(C A), ct(B)}, {on(C B), on(B A), ct(C)}, {on(A B), ct(A), ct(C)}, {on(A C), ct(A), ct(B)}, {on(B A), ct(B), ct(C)}, {on(C A), ct(B), ct(C)}, {on(C B), ct(A), ct(C)}, }

Current state: $S = \{on(A,B), on(B,C), ct(A)\}$

Put-operations:

name: *put*$_1$
pre : {cleartop(A), cleartop(B), ¬ on(A,C)}
op : put(A,B)
eff : ADD {on(A,B)} DEL {ct(B)}

name: *put*$_2$
pre : {cleartop(A), cleartop(B), on(A,C)}
op : put(A,B)
eff : ADD {ct(C), on(A,B)} DEL {on(A,C), ct(B)}

name: *put*$_3$
pre : {cleartop(B), cleartop(C), ¬ on(B,A)}
op : put(B,C)
eff : ADD {on(B,C)} DEL {ct(C)}

name: *put*$_4$
pre : {cleartop(B), cleartop(C), on(B,A)}
op : put(B,C)
eff : ADD {ct(A), on(B,C)} DEL {on(B,A), ct(C)}.

Calculating predecessors:

$$\begin{array}{rcl}
Res^{-1}(S, put_1) & = & \{on(B,C), ct(A), ct(B)\}\surd \\
Res^{-1}(S, put_2) & = & \{on(B,C), ct(A), on(A,C), ct(B)\} * \\
Res^{-1}(S, put_3) & = & \{on(A,B), ct(A), ct(B), ct(C)\} * \\
Res^{-1}(S, put_4) & = & \{on(A,B), ct(B), on(B,A), ct(C)\} * .
\end{array}$$

Figure 3.4: Calculating predecessor states and selecting admissible predecessors

where two blocks are lying on C simultaneously. There are also no state in $\mathcal{D}$ where A lies on B and B is cleartop (or where A lies on B and B simultaneously on A).

The crucial (and most time-consuming) aspect in backward-planning is to calculate the variable bindings [BW94] which are used to instantiate operators and thereby to generate predecessor states. As for determining admissible states, we can again rely on the predefined set of states $\mathcal{D}$.

Definition 8 (Determination of variable bindings). Let S be the current state. For each variant $\hat{o}$ of an operator with precondition φ_0 (which can either be $\varphi_0 \cup \varphi$ or $\varphi_0 \cup \neg\varphi$) and effect $\alpha\delta$ (which can either be $\alpha_t\delta_t$ or $\alpha_e\delta_e$) we determine the set of legal variable bindings in the following way:

1. $\Sigma = \emptyset$
2. Find all substitutions σ_i with $\alpha_\sigma \subseteq S$.
   Set Σ to $\Sigma = \{\sigma_i \mid i = 1 \ldots n\}$.
3. For each σ_i find all compatible substitutions ρ with $\varphi_{0_{\sigma_i\rho}} \subseteq S'$ with $S' \in \mathcal{D}$.
   Set each $\sigma_i \in \Sigma$ to $\sigma_i\rho$.

The composition of substitutions σ and ρ is compatible iff for $(v \leftarrow t) \in \sigma$ there exists no $(v' \leftarrow t') \in \rho$ with $v = v'$ and $t \neq t'$, where t, t' are constants.

We will show below that by using look-ups in the set of states $\mathcal{D}$ determining variable bindings can be performed much more efficient than in usual backward-planning approaches.

Finally, we can define the application of "parallel operations". That is, we describe how the complete set of legal predecessors of a state is calculated.

Definition 9 (Applying a set of operations backward). Let Res_p^{-1} be a function from states and sets of operations to sets of states $Res_p^{-1} : 2^P \times 2^O \rightarrow 2^{2^P}$ and $\mathcal{D}$ the set of all states of a given finite problem domain.

The backward application of a set of operations $\{o_1 \ldots o_n\}$ results in a set of admissible predecessor states $\{S_1 \ldots S_m\}$ with $m \leq n$ and is defined in the following way:

$$Res_p^{-1}(S, \{o\}) = Res^{-1}(S, o)$$

$$Res_p^{-1}(S, \{o_i \ldots o_n\}) = \begin{cases} \{Res_p^{-1}(S, o_1)\} \cup Res_p^{-1}(S, \{o_{i+1} \ldots o_n\}) & \\ \quad with\ i = 1 \ldots n & ;\ if\ Res_p^{-1}(S, o1) \in \mathcal{D} \\ Res_p^{-1}(S, \{o_{i+1} \ldots o_n\})\ with\ i = 1 \ldots n;\ otherwise. & \end{cases}$$

For the state $\{$*on(B,C), ct(B), ct(A)*$\}$ the application of a set of *put* operations results in the admissible predecessors $\{$ $\{$*ct(A), ct(B), ct(C)*$\}$, $\{$*on(B,A), ct(B), ct(C)*$\}\}$. Of course, when planning in the blocksworld domain we also regard applications of *puttable*-operations.

3.1.3 Implementation details

In our implementation, operators are defined in the following way:

```
(setq rules '(
  (rule puttable
    (if ( ct (> x) ) ( on (< x) (> y) ) )
    (then (puttable (< x)))
    (conq ( add ((ct (< y)))
            del ((on (< x) (< y))) ) )
  )
  (rule put
     (if ( ct (> x) ) ( ct (> y) ) )
     (then (put (< x) (< y) ) )
     (conq (cond ((on (< x) (> z))
                     (add ((on (< x) (< y)) (ct (< z)))
                      del ((ct (< y)) (on (< x) (< z)))))
                  ( T ( add ((on (< x) (< y)))
                        del ((ct (< y))) ) ) ) ) )
))
```

The representation of variables as `(> x)` or `(< y)` is different from the often used form `?x`. We were inspired by Winston's definition of match-select-apply cycles for production systems [WH89]. The symbol $>$ indicates that a variable has to be instantiated in accordance to the current state; the symbol $<$ that a variable has be instantiated in accordance with the current set of bindings. For backward-application of operations the semantics of $<$ rsp. $>$ is switched. In the next implementation we want to replace $>$ / $<$ by the usual ? notation. This can be realized with only a slight modification in the algorithm: If there is a variable: first look in the list of current bindings, if the variable is bound, take this value, otherwise, obtain the binding from the current state.

We already remarked above that we hope to expand our first implementation of DPlan to full ADL-syntax [Ped89]. Furthermore, we want to introduce the possibility of defining data types for variables and we want to allow for external function calls in operations similar to PRODIGY [BCE$^+$92]. These extensions are crucial if we want to use our planner for calculating initial programs for list and number problems which are usually regarded in program synthesis (see chapter 4 and appendix E).

3.2 The algorithm DPlan

After defining the backward application of sets of operations to a current state thereby generating the set of all admissible predecessor states we are ready to define the planning problem.

Definition 10 (Planning problem). A **planning problem** $\mathcal{P}(\mathcal{O}, \mathcal{D}, \mathcal{G})$ is a 3-tuple where $\mathcal{O}$ is the set of operators, $\mathcal{D}$ a set of initial states and G a set of planning goals. All states $S \in \mathcal{D}$ and G are sets of positive ground literals. In the context of our work, $\mathcal{D}$ is the set of all states of a finite problem domain.

Planning goals are currently restricted to conjunctions of (possibly interdependent) subgoals (cf. {*on(A,B), on(B,C)*}). We are planning to extend goal definitions to negation, disjunction and universal quantification in the future.

Let C be the set of all constants (i.e. arguments of predicates $P \in S$) occurring in $\mathcal{D}$. The set of operations O is the set of all possible ground instances of $\mathcal{O}$ with respect to C.

Definition 11 (Minimal spanning tree of $\mathcal{P}$). [3] The minimal spanning tree of a planning problem $\Theta = (V, E)$ with the set of nodes V, the set of edges $E \subseteq V \times V$, and two labeling functions $\beta_V : V \rightarrow \mathcal{D}$, $\beta_E : E \rightarrow O$ is a tree with arbitrary branching factor, i.e.

$$\Theta = \theta \begin{cases} \lambda & (empty\ tree) \\ v \in V & (leaf) \\ v(\theta_1 \ldots \theta_b) & with\ 0 \leq b \leq ||\mathcal{D}|| \end{cases}$$

for which the following conditions hold:
(With Θ_l we denote the tree from the root to level l and with N_l we denote all nodes in Θ_l.)

1. The root $\Theta(N, E)_0$ is
    - $\perp$ if there is no state $S \in \mathcal{D}$ for which $G \subseteq S$ holds (the planning problem is not solvable), or
    - $v\ with\ \beta_V(v) = S$ if $S \in \mathcal{D}$ and $G \subseteq S$, and if there is no other state $S' \in \mathcal{D}$ with $G \subseteq S'$ (there is exactly on state which fulfills the goal), or
    - (root node and first level of Θ) $v(w_1 \ldots w_n)$ with $\beta_V(v) = G$ and $\beta(w_i) = S$ if $S \in \mathcal{D}$ and $G \subseteq S$ and $\beta_E(v, w_i) = \epsilon$ for all w_i, $2 \leq i \leq ||\mathcal{D}||$ (there is more than one state fulfilling the goal).
2. For each leaf node v of Θ_l the children are all $w \in Res_p^{-1}(\beta_V(v), O)$ with w is admissible (i.e. $w \in \mathcal{D}$) and $w \notin N_l$. For each edge (v, w) we have the label $\beta_E(v, w) = o$ with $Res^{-1}(\beta_V(v), o) = w$.

For the formal background and a variety of algorithms for spanning trees, see cf. [GH85, Tar83].

Definition 11 entails the planning algorithm (which we already described informally in chapter 2) given in table 3.1[4]. We will discuss its soundness,

---

[3] Remark: Def. 11 and the presentation of the algorithm are a bit overloaded. I will formulate both "more elegantly" soon.

[4] The initialization and the tree expansion are not easy to read, because I discriminate between nodes v, w and node labels $\beta_V(v)$ which are corresponding to states. When simplifying def. 11 the algorithm can be simplified accordingly. Note, that the planning algorithm can be implemented more efficient when integrating the calculation of variable bindings and state candidates in one loop, see efficiency of DPlan below.

completeness and optimality in section 3.4 and we will give further examples below and in section 4. Information about the current implementation of DPlan is given in appendix A.

Note, that in the implementation of DPlan we generate a graphical output of Θ where the operations are not given as edge labels but as intermediate nodes between nodes representing problem states.

Illustration: a blocksworld example

To generate a plan for building a tower of blocks A, B, C in a three-block world, we present DPlan with the following problem specification:

```
(setq states '(
 ((on a b) (on b c) (ct a))
 ((on b c) (ct a) (ct b))
 ((ct a) (ct b) (ct c))
 ((on b a) (on a c) (ct b))
 ((on c a) (on a b) (ct c))
 ((on a c) (on c b) (ct a))
 ((on b c) (on c a) (ct b))
 ((on c b) (on b a) (ct c))
 ((on a b) (ct a) (ct c))
 ((on a c) (ct a) (ct b))
 ((on b a) (ct b) (ct c))
 ((on c a) (ct b) (ct c))
 ((on c b) (ct a) (ct c))
))

(setq goal
 '( (on a b) (on b c) )
)
```

The ordering of states and of goal propositions is arbitrary. The operator definitions for `put` and `puttable` were already presented in section 3.1.3.

Planning starts with checking whether there are states fulfilling the `goal`: state `((on a b) (on b c) (ct a))` fulfills the goal and no other state, therefore it gets the root of the plan and is deleted from the list of current states (see "initialization" in table 3.1).

In the next step (first node expansion, see table 3.1), DPlan generates all such instantiations of all operator variants which ADD-list literals match with a literal of the current state. That is, candidates are `(puttable x)` with `y = A` realizing `(ct A)`, `(put A B)` with and without `(on A z)` realizing `(on A B)` and `(put B C)` with and without `(on B z)` realizing `(on B C)`. Each operation candidate is applied backwards (adding literals from the DEL-list and preconditions and deleting literals from the ADD-list from the current state). The calculation of state candidates, selection of admissible candidates, and determination of variable bindings was already illustrated in section 3.1.2. In figure 3.4 we demonstrated for the `put` operator, that only one of the four candidates results in

Table 3.1: The planning algorithm DPlan

input : a planning problem $\mathcal{P}(\mathcal{O}, \mathcal{D}, G)$

output : the minimal spanning tree Θ for $\mathcal{P}$ or $\perp$

initialization: $\Delta = \mathcal{D}$, the set of current states

$$\Theta := \begin{cases} (1)\perp;\ \mathit{if\ there\ exists\ no\ state}\ S \in \mathcal{D}\ \mathit{with}\ G \subseteq S \\ (2) v\ \mathit{with}\ \beta_V(v) = S \in \mathcal{D}\ \mathit{and}\ G \subseteq S;\ \mathit{if\ there\ exists\ no\ other\ state} \\ \hfill S' \in \mathcal{D}\ \mathit{with}\ G \subseteq S' \\ (3) v(w_1 \ldots w_n)\ \mathit{with}\ \beta_V(v) = G\ \mathit{and}\ \beta_V(w_i) = S_i\ \ \forall S_i \in \mathcal{D}\ \mathit{and}\ G \subseteq S_i, \\ \mathit{with} \beta_E(G, S_i) = \epsilon, i = 1 \ldots n, 2 \leq n \leq \|\mathcal{D}\|;\ \mathit{otherwise} \end{cases}$$

for (2) and (3) Δ is reduced: $\Delta := \Delta\ \{\beta_V(v)\}$ for (2) and $\Delta := \Delta\ \{\beta_V(w_i)\} \forall w_i$ for 93).

- **Terminal condition:** $\Delta = \emptyset$ (success) or $\neg\exists v$ for which def. 11.3 holds (failure)
- **Tree expansion:** (1)
  If $\Theta = \perp$
  Then return bottom
  Else For all leafs v_i of $\Theta = \theta(v_1 \ldots v_n)$ do
  1. Calculate $\Omega = \{w | \beta_V(w) \in Res_p^{-1}(\beta_V(v), O) \in \Delta\}$
  2. $\Delta := \Delta \setminus \{\beta_V(w)\} forall w \in \Omega$
  3. Expand Θ:
     Replace leaf v_i by $v_i(w_1 \ldots w_m)$ with $\beta_E(v_i, w_j) = o$ with $Res_p^{-1}(\beta_V(v_i), O) = \{w_1 \ldots w_m\}$.

  endfor
- **Node expansion:** (2)
  For the current state $S = \beta_V(v)$ and for each variant $\hat{o}$ of an operator in $\mathcal{O}$ with $\hat{o} = \varphi_0, \alpha\delta$
  - Determination of all sets of legal bindings (2a)
    1. $\Sigma = \emptyset$
    2. Find all substitutions σ_i with $\alpha_\sigma \subseteq S$
    3. $\Sigma := \{\sigma_i \mid i = 1 \ldots k\}$
    4. For each σ_i find all compatible substitutions ρ with $\varphi_{0_{\sigma_i \rho}} \subseteq S'$ with $S' \in \mathcal{D}$.
    5. Set each $\sigma_i \in \Sigma$ to $\sigma_i := \sigma_i \rho$.
  - Construct all operation candidates $O = \{o \mid \hat{o}_{\sigma_i}\}$
  - Calculate all state candidates

    $$\mathcal{C} = Res^{-1}(S, o) = \begin{cases} (S \cup (D(S, o) \cup \varphi_0)) \setminus A(S, o) & ;\ \mathit{if}\ A(S, o) \subseteq S \\ \mathit{undefined\ and}\ O := O \setminus \{o\} & ;\ \mathit{otherwise} \end{cases}$$

    with $A(S, o)$ as instantiated ADD-list of o and $D(S, o)$ as instantiated DEL-list.
  - **For all** $c \in \mathcal{C}$ with $c \notin \Delta$ do $\mathcal{C} := \mathcal{C} \setminus c$
  - Return all pairs (o, S') with $S' = Res^{-1}(S, o) \in \mathcal{C}$

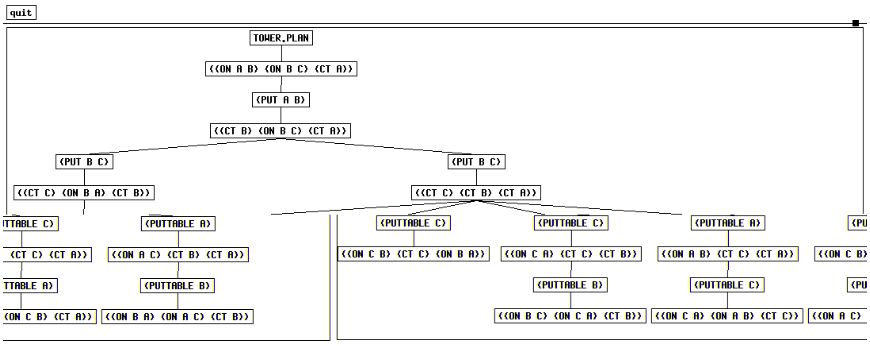


Figure 3.5: Result of DPlan for the tower problem

an admissible predecessor – `(put A B)` where `A` was not lying on another block.

We give an additional illustration for `puttable`: First we find the legal bindings $x = B$ and $x = C$ (see 2a in table 3.1) resulting in the operation candidates:

name: $puttable_1$
pre: $\{cleartop(B)\}$
op: $puttable(B)$
eff: $ADD\ \{cleartop(A)\}\ DEL\ \{on(B,A)\}$

name: $puttable_2$
pre: $\{cleartop(C)\}$
op: $puttable(C)$
eff: $ADD\ \{cleartop(A)\}\ DEL\ \{on(C,A)\}$

with the not admissible predecessors

$Res^{-1}(S, puttable_1) = \{on(A,B), on(B,C), on(B,A), ct(B)\}$
$Res^{-1}(S, puttable_2) = \{on(A,B), on(B,C), on(C,A), ct(C)\}$.

That is, the expansion of the root results in the application of only one operation `(put A B)` for the admissible predecessor state `((on B C) (ct A) (ct B))`. This state is deleted from the set of current states. Plan expansion proceeds for this single leaf which has two legal predecessors and so on. The final plan is given in figure 3.5. Note, that for initial states corresponding to towers whose blocks are sorted in reverse to the desired goal, the blocks have not to be unstacked completely. Instead the side-effect of *put* – clearing a block z if x lies on a block and not on the table – can be exploited for an optimal plan. While this plan is more intelligent than simply first putting all blocks on the table and then constructing the desired tower it complicates macro synthesis: reverse sorted towers has to be considered as exception and treated differently than all other input states.

Table 3.2: Plan execution as depth-first search

- Let I be the input state (initial state) and $L = ((\Theta_0))$ the list of paths (Θ_0 is the root of the plan).
- Until $L = (\ )$ or $Head(Head(L)) = I$ Do
  - Remove $L_a = Head(L)$ from L.
  - For $Head(L_a)$ find all immediate successors $S_i, i = 0 \ldots n$ in Θ with edges $(Head(L_a), S_i) = o_i$.
  - Create new lists $(S_i, o_i, Tail(L_a))$.
  - Insert the lists in front of $Tail(L)$.
- If $L = (\ )$
  Then announce failure (I is no admissible state in Θ)
  Else $\overline{Res}(I, Tail(Head(L)))$ (apply operation sequence)

3.3 Plan execution

The output of DPlan represents operation sequences for all possible states of a given domain. For example, the Dplan for the *tower* problem (see figure 3.5) contains the plans for building a tower A, B, C from initial state *on(C,B), on(B,A), ct(C)* as left most path, from initial state *on(B,C), ct(A), ct(B)* as partial path and so on.

Note that representing *action sequences* for each possible state differs from the state-action tables used in universal planning [Sch87, Sch95]. We can calculate action sequences because DPlan is restricted to deterministic worlds. That is, state changes can *only* occur as result to an operation application and there are no possible influences from other agents or the environment. Thus, it cannot be the case, that one of the planning objects is suddenly missing or is not at the expected position. With this restrictions, plan execution can be performed a simple tree-search algorithm. Both depth-first (see table 3.2) and breadth-first (see table 3.3) search are possible (we use a similar notation as [Win92]).

For the abstract tree given in figure 3.6, the depth-first algorithm generates the following sequence:

```
I = g, L = ((a))
L = ((b o1) (c o2) (d o3))
L = ((e o4 o1) (f o5 o1) (c o2) (d o3))
L = ((f o5 o1) (c o2) (d o3))
L = ((c o2) (d o3))
L = ((g o6 o2) (d o3))
-> apply <o6, o2>.
```

The corresponding sequence for the breadth-first algorithm is:

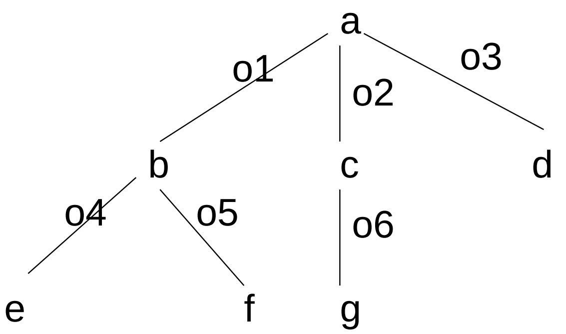


Figure 3.6: Abstract minimal spanning tree

Table 3.3: Plan execution as breath-first search

- Let I be the input state (initial state) and $L = ((\Theta_0))$ the list of paths (Θ_0 is the root of the plan).
- Until $L = (\ )$ or $Head(Head(L)) = I$ Do
  - Remove $L_a = Head(L)$ from L.
  - For $Head(L_a)$ find all immediate successors $S_i, i = 0 \ldots n$ in Θ with edges $(Head(L_a), S_i) = o_i$.
  - Create new lists $(S_i, o_i, Tail(L_a))$.
  - Insert the lists at the end of $Tail(L)$.
- If $L = (\ )$
  Then announce failure (I is no admissible state in Θ)
  Else $\overline{Res}(I, Tail(Head(L)))$ (apply operation sequence)

```
I = g, L = ((a))
L = ((b o1) (c o2) (d o3))
L = ((c o2) (d o3) (e o4 o1) (f o5 o1))
L = ((d o3) (e o4 o1) (f o5 o1) (g o6 o2))
L = ((e o4 o1) (f o5 o1) (g o6 o2))
L = ((f o5 o1) (g o6 o2))
L = ((g o6 o2))
-> apply <o6, o2>.
```

Alternatively, we could decompose the plan tree and save a state/action-sequence table (as `(a nil)`, `b o1`, … `(e o4-o1)`, …). Searching the table has the same complexity as search in the tree (if we do not introduce a efficient hashtable representation mechanism).

3.4 Termination, Soundness, Completeness, Optimality, Efficiency

As usual, we can give no guarantee that the given specification of a problem (i.e. goal, operators and set of states) are sound and complete. The following theorems and proofs address the characteristics of DPlan with respect to a given problem specification.

Theorem 1 (Termination). *For given planning problem $\mathcal{P}(\mathcal{O}, \mathcal{D}, \mathcal{G}$ DPlan terminates either successful and returns the minimal spanning tree Θ of $\mathcal{P}$ or it terminates unsuccessful and returns $\perp$.*

> **Proof:**
> (1) First we proof that DPlan detects the unsolvability of a planning problem (termination with $\perp$). There are two cases of unsolvability:
>
> - Total unsolvability: There is no state $S \in \mathcal{D}$ with $G \subseteq S$.
> - Partial unsolvability: The set of states $\Delta \subseteq \mathcal{D}$ which are not yet inserted in Θ is not empty but there exists no current leaf v in Θ for which $Res^{-1}(v, o)$ returns an admissible state in Δ.
>
> Total unsolvability is detected *before* the entrance in the planning loop (see table 1, tree expansion) and DPlan terminates immediately. In the case of partial unsolvability we have a current tree $\Theta_t = v(v_1 \ldots v_n)$ with leafs $v_i, i = 1 \ldots n$. If $Res_p^{-1}(v_i, O) = \emptyset$ then DPlan tries to expand v_{i+1}. If $Res_p^{-1}(v_i, O) = \emptyset$ for all v_i than DPLan terminates with $\Theta_t = Theta_{t+n}$ and returns $\perp$.
>
> (2) DPlan terminates successfully if all states in $\mathcal{D}$ are regarded in Θ, i.e. $\Delta = \emptyset$ (this is a termination condition for the planning loop, see table 1)[5]. *q.e.d.*

---

[5] Because partial solutions might also be of interest, in the implemented version DPlan returns the partial spanning tree but without announcing success.

Theorem 2 (Soundness). [6] *If DPlan returns a minimal spanning tree* Θ *for a given planning problem* $\mathcal{P}(\mathcal{O}, \mathcal{D}, \mathcal{G}$ *then each path from an arbitrary node* $S \in$ *Theta to the root is a operation sequence* $\mathcal{Q} =< o_i >$ *with* $0 \leq i \leq n$ *with* $G \subseteq \overline{Res}(S_i, < o_i >)$*, i.e.* $\mathcal{Q}$ *is a solution.*

Proof:
We prove soundness by induction over the length of operation sequences relying on definitions 11, 9 and 5.
[base case] $\mathcal{Q} =<>$
We have to show that $\mathcal{Q} =<>$ holds exactly for such states S with $G \subseteq S$. In definition 11 no operations are introduced only if $\mathcal{P}$ is totally unsolvable (but then $\Theta = \perp$), or, if there exists a unique state S with $G \subseteq S$, or if there exists a set of states S with $G \subseteq S$. So for all cases were $\Theta \neq \perp$, the empty sequence of operations is only valid for the root of Θ as defined in definition 11.1.
[induction step] $\mathcal{Q} =< o_1 \ldots o_n >$
We regard an arbitrary path from a node $S \in \Theta$ to the root of Θ with edge labels $< o_n \ldots o_1 >$. We assume that the induction hypothesis holds for this path, i.e., $G \subseteq \overline{Res}(S, < o_n \ldots o_1 >$. During planning, we calculate all admissible predecessors of S not yet regarded in Θ. Applying $Res_p^{-1}(S, O)$ can have the following results:

1. $Res_p^{-1}(S, O) = \emptyset$
2. $Res_p^{-1}(S, O) = S'$ with $o_{n+1} = (S', S)$
3. $Res_p^{-1}(S, O) = S'_1 \ldots S'_m$ with $\{o_{n+1_i} \mid o_{n+1} = (S'_i, S) \forall S_i \in Res_p^{-1}(S, O)\}$.

For case (1) we do not introduce a new node (and thereby also no new operation), that is $G \subseteq \overline{Res}(S, < o_n \ldots o_1 >$ still holds. For case (2) we have an admissible predecessor of S in accordance to definition 9. That is, $Res(S', o_{n+1}) = S$ and therefore, in accordance to definition 5, $G \subseteq \overline{Res}(S', < o_{n+1}, o_n \ldots o_1 >)$. For case (3) we have a set of admissible predecessors of S in accordance to definition 9. That is, $Res(S'_i, o_{n+1_i}) = S$ for all $S'_i \in Res_p^{-1}(S, O)$. Therefore, in accordance to definition 5, $G \subseteq \overline{Res}(S'_i, < o_{n+1_i}, o_n \ldots o_1 >)$ holds for all S_i. *q.e.d.*

Theorem 3 (Completeness). *Completeness follows from soundness and termination.*

Theorem 4 (Optimality). *For a planning problem* $\mathcal{P}(\mathcal{O}, \mathcal{D}, \mathcal{G}$ *the resulting plan* Θ *is a minimal spanning tree for problem space* $\mathcal{P}$ *and it represents the minimal sequence of operations for transforming each state* $S \in \mathcal{D}$ *into a state fulfilling the goal. That is,* $G \subseteq \overline{Res}(S, < o_1 \ldots o_n >) \forall S \in \mathcal{D}$ *and* $\neg \exists < o'_1 \ldots o'm >$ *with* $G \subseteq \overline{Res}(S, < o'_1 \ldots o'_m >)$ *with* $m < n$*. In other words, Theta is a spanning tree with minimal depth for* $\mathcal{P}$*.*

---

[6] Remark: The proof could be more compact if I introduce a definition for the relation between paths in Θ and forward-application of operation sequences.

Proof:
We can proof this theorem by structural induction over the levels of Θ.

[base case:] $\Theta = \Theta_0$
We have to show, that the root Θ_0 of Θ represents all states $S \in \mathcal{D}$ which already fulfill G.
This follows immediately from definition 11: All states $S \in \mathcal{D} with G \subseteq S$ are regarded when introducing the root of Θ. If there is more than one state fulfilling $G \subseteq S$, a "dummy"-root is introduced with these states as immediate successor nodes and empty edge labels.

[induction step:] $\Theta = \Theta_l$
We regard a partial spanning tree, expanded to depth l covering a subsets of states from $\mathcal{D}$ for which the induction hypothesis holds. That is, for all leafs $S \in \Theta_l$ holds $S \in \mathcal{D}$ and $G \subseteq \overline{Res}(S, < o_1 \dots o_n >)$ and $\neg\exists < o'_1 \dots o'm >$ with $G \subseteq \overline{Res}(S, < o'_1 \dots o'_m >)$ with $m < n$. The expansion of the nodes S follows definition 11. That is, an admissible predecessor state $S' = Res^{-1}(S, O)$ is only then introduced into Θ if $S' \notin \Theta_l$, otherwise S' is a node in $\Theta_k, k \leq l + 1$ with $G \subseteq \overline{Res}(S, < o_1 \dots o_k >)$. *q.e.d.*

Theorem 5 (Efficiency). *DPlan calculates the minimal spanning tree Θ with a worst case effort of $O(n^2)$*[7].

Proof:[8]
The expansion of Θ (outer loop, (1) in table 1) needs $(n-1)$ cycles in the worst case. That is, each node has exactly one predecessor. So, in each cycle only one node is removed from Δ.
The expansion of a node in Θ (inner loop, (2) in table 1) needs in the worst case $||\Delta||$ cycles. That is, for calculating all sets of possible bindings and determining the admissibility of predecessors, each state which still has to be regarded has to be checked.
So, for $||\mathcal{D}|| = n$ we have in the worst case $n + n - 1 + n - 2 \dots 1 = \sum_{i=1}^{n} i = \frac{n(n-1)}{2} \approx n^2$ steps. *q.e.d.*

We want to conclude this section with some remarks regarding efficiency of DPlan. The complexity $O(n^2)$ is not too bad for a backward planning system which additionally regards not a single, but a complete set of initial states. The efficiency gains in comparison with other backward planners (cf. TOPI, [BW94]) is mainly due to the fact that we can use look-ups to a (stepwise getting smaller) set of states for calculating variable bindings (which has complexity $||I||^{||G||}$ with $||I||$ as number of literals in the initial state and $||G||$ as number

---

[7] This holds for an optimized version of the algorithm given in table 1 and the current implementation where all calculations performed after step 4 of calculating the bindings (in expand node) are also performed in step 4.

[8] Remark: currently very quick and dirty! Especially the aspect of calculating sets of bindings should be checked again.

of goal literals for TOPI, that is, is NP-hard in the worst case[9]). Furthermore, by calculating a minimal spanning tree, DPlan works completely without backtracking. Backtracking can in the worst case result in calculating the complete search tree. For a problem domain with n states and m operations, the worst case complexity is n^m. Finally, generation of optimal (i.e. shortest) plans can for ADL-planners only be guaranteed by using a breadth-first strategy (for example, IPP [KNH97] cannot guarantee optimality). DPlan uses a kind of optimized breadth-first search – where the number of state candidates shrinks at each level. Finally, we propose that our strategy of storing all (not yet regarded) alternative operations and their outcomes, could be useful also in the context of other planning systems: Usually, for each planning step all alternative actions are calculated, but after selecting one action the other ones are forgotten. Thus, in case of backtracking, all options have to be calculated again.

One might argue that we load the greater part of effort on the users of DPlan who have to specify all states of a problem. We have some arguments against this proposition: (1) Remember, that our main goal is providing initial programs for program synthesis or – taking a planning perspective – generalizing cyclic macro-operations from planning. If providing a planning system with such experience that a more general strategy can be inferred, a traditional planner might be called eventually with all possible states of a given problem as initial state. The effort of planning immediately for n states with the method used by DPLan is sure more efficient than calling a standard planning system (with backtracking) n times. An argument agains our strategy is, that we cannot model incremental learning as in PRODIGY [VCP$^+$95]. (2) DPlan is not intended primarily as efficient planning system but as a method for generating initial programs. We argue, that we can bridge the gap between universal planning and program synthesis by planning only for problems with small complexity and than generalize over the structure of these problems (for example from a Tower of Hanoi problem with 3 discs to problems with an arbitrary number of discs). (3) Systems which use domain axioms to guarantee the generation of valid plans – as model-based planning systems [CRT98] – make the task of domain and problem specification much harder for the user than DPlan. For the standard user it is clearly more easy to enumerate all possible states of a small example domain than to formulate a complete and consistent set of axioms.

To take the load of enumerating all states of a problem from the user, we could use one of the following two strategies: (a) generate the states automatically from an initial state using the following algorithmical idea: generate the solution path from initial state to goal by forward-application of rules. For all states lying on the solution path which are not yet regarded: take these states as new initial states and calculate their solution path. Terminate if no solution path contains states which were not yet regarded[10]; or (b) only provide DPlan with an goal state and with the set of objects (for example blocks A, B, C) to be considered, calculate predecessors as specified in table 1 but with the fol-

---

[9] Remark: see theorem 1 in [BF97]: for operators with a fixed upper number of parameters k, time effort is polynomial in k.

[10] Remark: This is a first idea and not really thought through.

lowing modifications: generate all operation candidates by instantiate operators with all predefined objects and check admissibility of a new state S' by forward application $Res(o, S') = S$ for the current state S.

3.5 Comparison with Other Planners

To discuss DPlan in relation to other planning systems, we first introduce some classification criteria [RN95, AHE90, Wel99b]:

- **state space vs. plan space**. Planning can either be described as search in state space or as search in the space of partial plans. State based planners introduce new (intermediate) states by applying (instantiated) operators to a current state. Planning terminates if an operator sequence transforming the initial state into the goal state is found. That is, a plan is a path though the search tree with states as nodes and operators as arcs. Plan space planners search though the space of (partial) plans. An initial plan ("null plan") consists of the initial state – as preconditions which are initially true – and the goal statement – as postconditions (effects) which finally must be true. A partial plan is expanded by refinement operators and modification operators: Refinement operators add ordering constraints (putting one plan step before another) and binding constraints for variables, modification operators introduce new planning steps. Planning terminates if the plan is complete (every preconditions of every step is achieved by some other step) and consistent (there are no contradictions in ordering and binding constraints).

- **progression vs. regression**. Progression planners search forward from the initial state to the goal; regression planners search backwards – backward planning does not rely on a complete description of states. The notion of forward/backward search is not strictly applicable to plan-spaced planners.

- **linear vs. non-linear**. Linear plans do not allow for interleaving of goals. That is, if a subgoal is selected, first all the steps to solve this subgoal are calculated after the next subgoal is considered. Linear planners are not complete (they can for example not, or not efficiently, solve the Sussman anomaly [Sus75])!

- **total vs. partial order**. Total order planners return a list (sequence) of planning steps; partial order planners allow to leave (independent) planning steps unordered.

- **domain and problem specification**. A problem is usually presented by an initial state and a (set/conjunction) of problem solving goals. The specification of operators is sometimes called domain theory. In **STRIPS** [FN71b] (initial) states are represented as conjunctions of function-free ground literals, goals are represented as conjunctions of function-free literals (which can contain implicitly existentially quantified variables). State descriptions must not be complete. Operators are represented by preconditions, an action description (name), and effects. Preconditions are a conjunction of atoms, effects conjunctions of literals. Effects are represented as ADD- and DEL- lists (some times as a single list of literals with the DEL-effects negated). Instantiated operators are called *actions*. Operators containing variables are called *operator schemes*. If all variables of an operator can be instantiated in a situation s in such a way that all preconditions are true, the operator is called *applicable*. **Situation calculus** [McC98] allows to represented problems and operators using the full expressiveness of first-order logic. All predicates (as part of state, goal and action descriptions) have a situation variable as (additional) argument. Currently, most planning systems use (at least a subset of) *ADL* (action description language) [Ped89] to represent operators. Using ADL gives us most of the expressiveness of situation calculus without sacrificing the efficiency of using STRIPS operators. ADL allows for conditional effects (eliminating premature commitment to an operator as source of inefficiency), negated and disjunctive goals, universal quantification. Additionally, updating of variable values and application of functions is allowed.

We can now classify some well-known planning systems:

- Classic work in planning is:
  - GPS [NS61]: a state-space, linear, total-order planner; it uses means-end analysis, starting with the planning-goals, attacking one goal after the other in arbitrary order, using depth-first search with backtracking.
  - QA3 [Gre69]: models planning as theorem proving, using situation calculus. Another prominent approach to planning as deductive proof problem was introduced by [MW87].
  - STRIPS [FN71b]: a state-space, linear, total order progression planner; a similar approach is realized in HACKER [Sus75].
  - NOAH [Sac75]: was the first partial-order planner, working in plan-space and being non-linear (i.e. allows interleaving of goals)[11].
  - [Wal75]: introduced goal-regression planning; their approach is based on state-space and generates non-linear and total order plans; a system in this tradition is for example INTERPLAN [Tat75].
- The next generation of planners:

---

[11] NOAH was originally characterized as hierarchical planner. Today, hierarchical planning describes – knowledge intensive – approaches which are based on operators on different levels of abstraction, as for example ABSTRIPS [Sac74] and hierarchical task network planners [ea94]

  - In the eighties and early nineties the main focus was on *plan-space partial-order planning*. Prominent examples are: TWEAK [Cha87], SNLP [MR91] and finally UCPOP [PW92], which is an extension of SNLP to ADL; see [Wel94] for an overview. In contrast to state-space planners, these planners employ a *least committment* strategy: that is, instantiations of variables and orderings of steps are delayed as long as possible. The central – and computationally most complex – aspect of these planners is to detect threads and calculate causal links between planning steps.
  - PRODIGY [Vel94] is a state-space, regression, non-linear, total order planner with a powerful domain specification language (cf. allowing type-hierarchies and infinite datatypes for variables). In contrast to the systems named above, the work in PRODIGY does not focus on efficiency but on combining planning and learning (allowing variable control strategies and learning of such strategies).
  - Additionally to these – so-called classic – planning systems, approaches for dealing with uncertain and/or incomplete information were developed (see [RN95], Chap. 13 for an overview). *Replanning* systems monitor plan execution, detect violations of current preconditions and generate a new plan starting from the current state. *Conditional planners* (cf. CNLP [PS92], B-PRODIGY [BV97]) introduce different subplans for different contexts (i.e. generate trees of plans). *Reactive planners*, as for example *universal planning* [Sch87], calculate the desired action for each possible state of the world (which is of course only possible in – small – finite domains). An alternative to planning in uncertain environments is the use of reinforcement learning.

- Current trends: A variety of new planning approaches attacking the problem of planning efficiency are developed. There is a renaissance of state-based approaches.
  - GRAPHPLAN [BF97, KNH97]: started the series of approaches which transfer techniques from other domains of computer science to planning. The concept of minimal flow in networks was applied to planning problems: In a first step, a planning graph is constructed by forward search, in a second step a plan is extracted by backward search. The planning graph is organized in levels corresponding to time steps. Nodes are facts (single ground literals in contrast to states) and actions. The first level represents all facts which are true in the initial state; the next level represents all actions which have their preconditions fulfilled by the initial facts and "no-op" actions (doing nothing preserves facts); the next level contains all add- and del-effects of the actions and the preserved facts. *Edges* are introduced only between immediate succeeding levels: preconditions edges from facts to actions and add- and del-edges from actions to facts. Additionally, a (incomplete) set of exclusion relations among

planning graph nodes is calculated (mutex) for interfering or competing actions and propositions. If all goal propositions are reached by add-edges and are not mutex at one level, plan expansion terminates with success and we know, that a valid plan exists for the given problem. In the next step, the planning graph is searched backward for a valid plan (a "flow" from the last level to the initial propositions). That is, a valid plan is a subgraph of the planning graph. Independent actions are not ordered but returned as a set if they occur at one level, that is, a partial order plan is returned. The original algorithm was based on STRIPS-like operators [BF97]; the planner IPP [KNH97] is an extension to an ADL subset. To use our terminology from above: graphplan approaches are state-based, nonlinear, partial order regression planners.

- SAT-planners, as BLACKBOX [KS98]: argue that planning is a special kind of theorem-proving and that first-order theorem-proving does not scale well. They propose to model planning in the framework of propositional satisfiability testing rather than to use specialized planning algorithms. Similar to GRAPHPLAN, SAT-planning relies on a propositional planning graph. States and actions are represented as (propositional) logical clauses. A plan corresponds to a model (i.e. truth-assignment) that satisfies a set of logical constraints representing initial state, goal state and domain axioms. Domain axioms characterize the consistency of states and legal state transitions (i.e. operator applications). Currently, different representations for planning graphs are investigated, for example graphplan-based encodings and state-based encodings. Representing states rather than single literals as nodes seems most promising. An efficient (but incomplete) method for finding plans is to use local stochastic search (as Walksat).
- Model-based planners: are a current approach, especially for planning in non-deterministic domains. Model-based planners generate universal plans in an reasonably efficient way by encoding plans as OBDD's (ordered binary decision diagrams) and using symbolic model checking to explore large state spaces [CRT98]. Application of symbolic model checking to deterministic planning problems is reported in [DGR98].

DPlan is a state-based non-linear total order regression planner. Similar to universal, conditional and model-based planners it does not rely on the presentation of an initial state but works on *sets of* states instead. In contrast to model-based planners but similar to conditional planners, we specify operators in STRIPS or a subset of ADL (and not predefine all possible actions) and we do not rely on domain axioms. Currently, domain specifications for DPlan allow for conditional effects, but we want to extend it to a larger subset of ADL.

Our planner is sound and complete and termination (with a valid plan or a partial plan) is guaranteed (which is not in general true for plan-based partial-order planners).

Planning effort is polynomial in the number of states. This statement is somewhat unfair when comparing DPlan with classical planners, because we do not consider the extraction of a single plan (for a given initial state) in our analysis (see section 3.3, plan execution). In a loose way, the (backward) generation of a plan in DPlan corresponds to the (forward) construction of a planning graph, which can also be done in polynomial time.

Finally, DPlan can guarantee to calculate optimal plans. This is true, because our planning algorithm is basically a breadth-first search.

Other than all planners mentioned above, the main motivation for DPlan was not to develop a sound, complete and efficient planner but to provide initial programs for program synthesis (see section 2). It is recognized in the planning community that there is a close relation between conditional planning and program synthesis (see [RN95], Chap. 13, p. 411). Our main interest is to cross-fertilize both fields: using planing methods to provide initial programs for program synthesis and using program synthesis to provide a technique for learning cyclic macro-operators – thereby making universal planning more efficient.

3.6 Empirical evaluation

Although it is not really meaningful to compare running times of planning algorithms which are implemented in different programming languages (LISP for PRODIGY, UCPOP and DPLAN; C for GRAPHPLAN and IPP) and using different amounts of information (single initial states vs. sets of states for DPlan and universal planners; domain axioms for model-based approaches[12]) we are planning to run experiments using the domains of the AIPS-98 planning competition (see `http://www.cs.cmu.edu/~ aips98/`, and the artificial domains from [BW94] – see also empirical results in [BF97, KNH97])[13]. While the comparison of the absolute values for performance does not give any valid information, a comparison of the shape of the problem size/time plots can be of interest. In particular, it should support the theoretical analysis of planning efficiency given above and give empirical insight in the scope of problems solvable by DPlan. Furthermore, we will use these planning domains to identify and illustrate meaningful possibilities for cyclic macro generation.

---

[12] We can ignore the use of domain-specific control knowledge cf. for PRODIGY.

[13] Remark: Hopefully, the comparison will be supported by participants of our student project in summer term 99.

Chapter 4

Transformation of Plans to Programs – Bridging the gap between planning and program synthesis

As described in chapter 2, our program synthesis system constructs cyclic macros by generalizing over unifiable substructures of a initial program. To provide an input to the synthesis system, the original output of DPlan has to be transformed into the format of such initial programs. Transformation of the original DPLan output – the minimal spanning tree – currently is fully implemented for linear plans and partially for plan-trees (for example the *tower* problem presented at the end of section 3.2, see figure 3.5). As described in chapter 3, the current implementation of DPlan cannot deal with universal quantification, infinite types and application of functions in operators. Therefore, planning for list and number problems is currently only possible in a limited way.

Because transformation of plans into initial programs is still work in progress, we will present our approach only informally and give examples together with an identification of open problems in plan transformation instead. An informal algorithm for transforming D-plans into initial programs is given in table 4.1. The motivation for each transformation step and an illustration for the *clear-block* problem was given in chapter 2. Note that our idea to linearize plans – unification of branches with identical operations and calculating intersections for conjunctions of predicate – corresponds to concepts introduced in inductive logic programming [LD94] and explanation based learning [MKKC86, BV96]. Furthermore, dealing with a sequence of different states in one path corresponds roughly to the concept of subsumption in ILP.

Table 4.1: Informal algorithm for transforming D-plans into initial programs (Steps marked with (*) are under investigation, that is, currently it is not proven that these steps preserve the semantic of a D-plan.)

1. If the plan is linear then
   - (a) **Rewrite Constants.** Replace constants by constructive expressions: recursively apply the rewrite-rules given in the domain specification to D-plan.
   - (b) **Determine Relevant Predicates.** Only keep such predicates of a state which occur in the ADD-list of the backward applied operation for each state-node in D-plan.
   - (c) **Generate binary tree.** For each predicate-node introduce a left successor *s* covering the case that the predicates are true. If no further operation is given, terminate the path with an asterix as last right successor of a predicate-node.
   - (d) **Rewrite into program term.** Transform the plan into the syntax of program terms (introduce conditional expressions).
2. else
   - (a) For each linear subplan: **Rewrite Constants.**
   - (b) **Determine Relevant Predicates.**
   - (c) **Unify.** branches with identical operations and replace the preceding predicate-nodes by their intersection. (*)
   - (d) If the plan is now linear
     then **Generate binary tree.**
     else For each linear subplan: **Generate binary tree.** (*)
   - (e) **Rewrite into program term.**

Table 4.2: Linear recursive macro-operations (here represented in LISP syntax)

```
; clear a block x: puttable all block topof x
(defun clearblock (x s) = (cond ((cleartop x) s)
                                (T (puttable (topof x) (clearblock (topof x) s)))))
; build a tower: put next block on current topof tower -- i.e. specified base
; (baseof x) has to be guaranteed to be cleartop!
(defun putblocks (x tw) = (cond ((on x (baseof s)) tw)
                                (T (put x (putblocks (baseof x) tw)))))
; unload objects o from vehicle v; (empty o) indicates that there are
; no more objects at the current position
(defun unloados (o d) = (cond ((at o d) d)
                              (T (unload o (unloados (next o) d)))))
; load objects o into vehicle v; assumes infinite capacity for v
(defun loados (o v)  = (cond ((insideR o) v)
                             (T (load o (loados (next o) v)))))
```

4.1 Inferring linear-recursive macro-operators

If the minimal spanning tree of a problem domain results in an linear order of states and involves only one operator, the plan can easily generalized to a cyclic macro-operation corresponding to a linear recursion. A typical example for a linear macro is the *clearblock* function discussed in chapter 2.

In general, linear recursive macros can be generalized from and applied to all problem domains which involve the repeated application of an operation to a data structure which is reduced by each application step. For the *clearblock* problem the operation *puttable* has to applied for each block lying on top of the block which has to be cleared. Other examples for linear domains are: building a tower by putting one block upon another block if the blocks currently considered are guaranteed to be clear, and loading or unloading a series of objects in or from a transportation vehicle (rocket, plane, truck, train, or briefcase). A sample of linear recursive macros is given in table 4.2.

4.1.1 Unstacking and building a tower

The domain specification for building a tower is given in table 4.3. The original output of DPlan is given in figure 4.1. Note, that we defined *put* with an conditional effect as above. But for the given problem domain the case that x is lying on another block (and not on the table) never occurs.

Plan transformation into an initial program starts with replacing constants by constructive expressions. We apply a rewrite-rule converse to *clearblock* – the block which must be put before another block in accordance with the goal statement is the *baseof* this block (see figure 4.2). The *put* operator adds $on(x, y)$ as primary effect – for this simple linear domain the side effect $ct(z)$ is never produced. Thus, the relevant predicate at each state node is $on(x, y)$ only. Figure 4.3 presents the binary planning tree for *putblocks*. In this transformation step, the leaf node is not reduced (see arguments and heuristics in section 2.4).

The binary tree can now easily be transformed into a program term (initial

Table 4.3: Domain specification for stacking blocks (*putblocks*)

```
; Linear put (analogous to clearblock)
; putting one block upon another when each block is clear when considered
; ===================================================================
(Make-Package 'pd-put)
(Export '(states goal rules is-c-pred c-eq ))
; ===================================================================
(setq states '(
 ((on a b) (on b c) (ct a))
 ((on b c) (ct a) (ct b))
 ((ct a) (ct b) (ct c))
))

(setq goal '( (on a b) (on b c) ) )

(setq rules
  '( (rule put
        (if ( ct (> x) )
            ( ct (> y) ) )
        (then (put (< x) (< y) ) )
        (conq (cond ((on (< x) (> z)) (add ((on (< x) (< y)) (ct (< z)))
                                       del ((ct (< y)) (on (< x) (< z)))))
                    ( T ( add ((on (< x) (< y)))
                          del ((ct (< y))) ) )
        )     )
     )
))
; ----------------------------------------------------------------
; Predicates which indicate constructive rewriting
(defun is-c-pred (p-name)
  (equal p-name 'on)
)
; (on x y) == [y = (next x)]
(defun c-eq (p)
  (list (nth 2 p) (list 'baseof (nth 1 p))) ; assoc-list
)
```

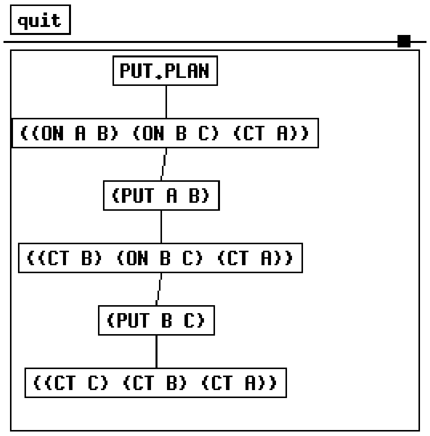


Figure 4.1: Output of DPlan for *putblocks*

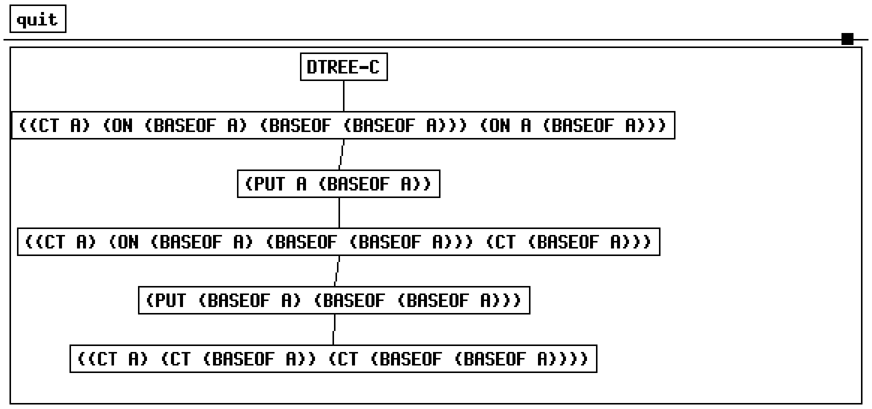


Figure 4.2: Rewritten constants for *putblocks*

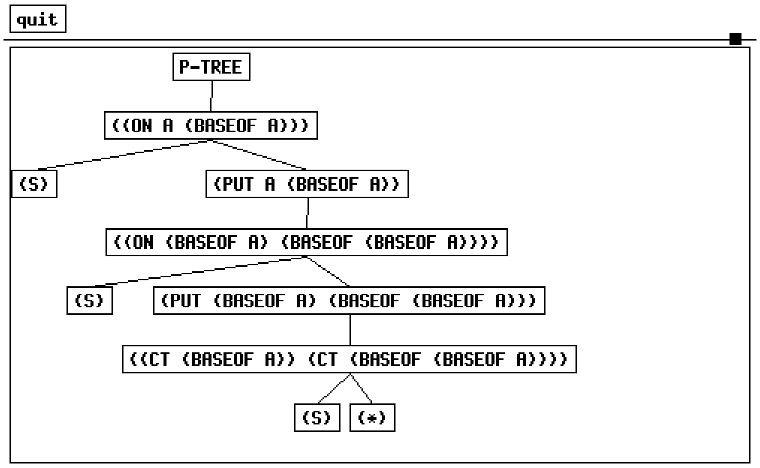


Figure 4.3: Binary tree for *putblocks*

program) by interpreting predicate nodes as conditions, left branches as then- and right branches as else-case:

$$
\begin{aligned}
G \;=\;\; & g(on(x, baseof(x)), s, put(x, \\
& \quad g(on(baseof(x), baseof(baseof(x))), s, put(baseof(x), \\
& \quad\quad \Omega))))
\end{aligned}
$$

or in LISP syntax

$$
\begin{aligned}
G \;=\;\; & (cond((on\ x\ (baseof))\ s) \\
& \quad (T(put\ x \\
& \quad\quad (cond((on(baseof\ x)(baseof(baseof\ x)))\ s) \\
& \quad\quad\quad (T(put(baseof\ x) \\
& \quad\quad\quad\quad omega)))))).
\end{aligned}
$$

The initial program can be folded in the recursive macro given in table 4.2. Note, that we named s into tw (for "current tower") for more intuitive readability. Remember, that the situation variable s is introduced to represent the current situation (state) which is returned if the predicates at the parent node are true in this state.

4.1.2 Unloading and loading objects

Veloso [VC93a] proposed the rocket and a more general transportation domain. Transportation problems typically involve the loading of objects into a transport-vehicle and their unloading at a given destination. Plan construction for transportation problems often relies on interleaving of subgoals to generate optimal plans – first loading *all* objects at the current position and then driving to the destination instead of loading one objects, driving to the destination, driving back, transferring the next object and so on. If resources (fuel) have to be taken into regard or if a transport vehicle can only drive in one direction as in the rocket domain, handling all objects at a location even becomes necessary for construction of valid plans. Applying *load* and *unload* macros in these domains could reduce planning effort dramatically.

In the next section we will see, that – although rocket is a relatively simple structured domain – it is not strictly linear when planning without restricting the order of handling objects. The rocket domain involves the execution of two consecuting linear macros *load* and *unload* and of an *move* operator inbetween. In the following we will deal with reduced problems: planning to load or unload objects from a vehicle. The problem specifications are given in tables 4.4 and 4.5. Because we will discuss macro generation for the rocket domain in the next section, the predicate names and rule definitions are taken from this domain. Note, that we do not consider the state *((insideR o2) (at o1 B) (atR B))* for *unload* and the state *((insideR o2) (at o1 A) (atR A))* for *load*. That is, we already imply an ordering of objects in the state definition. Otherwise, the resulting plan would not be linear. Furthermore, the rewrite rules for *load* and *unload* are converse (as for *clearblock* and *putblocks*) to guarantee that the basic case (the "bottom" object, which is not rewritten constructively) will appear in the root of the linear plan.

The DPlan outputs for *unload* and *load* are given in figures 4.4 and 4.5. The binary trees are given in figures 4.6 and 4.7. The resulting program terms are:

$$\begin{aligned} G \quad = \quad & g(at(o1B), B, unload(o1, \\ & \qquad g(at(next(o1)B), B, unload(next(o1), \\ & \qquad\quad \Omega)))) \end{aligned}$$

for *unload* and

$$\begin{aligned} G \quad = \quad & g(insideR(o2), V, load(o2, \\ & \qquad g(insideR(next(o2)), V, load(next(o2), \\ & \qquad\quad \Omega)))) \end{aligned}$$

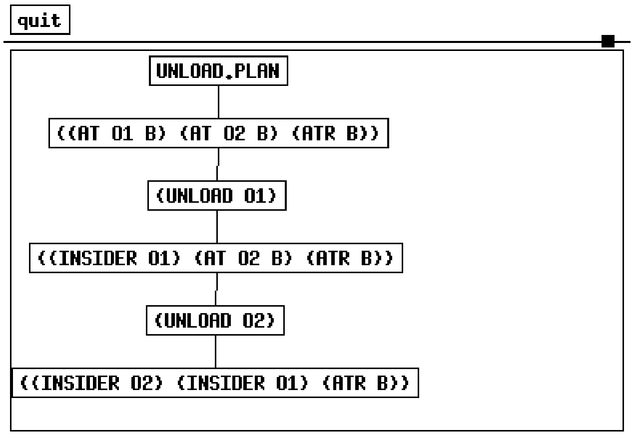
for *load*.

Table 4.4: Domain specification for unloading objects

```
; unload (basic macro for transportation domains)
; ====================================================================
; unload all objects from a vehicle
; --------------------------------------------------------------------
(Make-Package 'pd-unload)
(Export '(states goal rules is-c-pred c-eq))
; ====================================================================
(setq states '(
  ((at o1 B) (at o2 B) (atR B))
  ((insideR o1) (at o2 B) (atR B))
  ((insideR o1) (insideR o2) (atR B))
))

(setq goal  '((at o1 B) (at o2 B)))

(setq rules
  '( (rule unload
          (if (insideR (> x)) (atR (> y)))
          (then (unload (< x)))
          (conq (add ((at (< x) (< y)))
                 del ((insideR (< x)))
          )      )
      )
))
; ------------------------------------------------------------------
; Predicates which indicate constructive rewriting
(defun is-c-pred (p-name)
  (or (equal p-name 'at) (equal p-name 'insideR))
)
; (at o1 x) and (at o2 x) => o2 = next(o1)
; (insideR o1) and (insideR o2) => o2 = next(o1)
(defun c-eq (p)
 (cond ((equal (nth 1 p) 'o2)
         (list (nth 1 p) '(next o1))  ; assoc-list (o2 (next o1))
        )
))
```

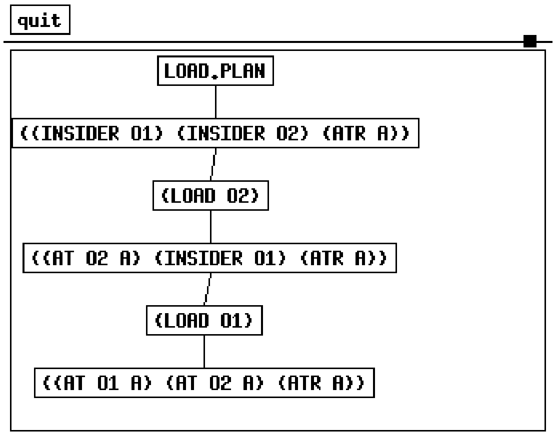
Table 4.5: Domain specification for loading objects

```
; load (basic macro for transportation domains)
; ================================================================
; load all objects from a vehicle (assuming a vehicle with infinite
; capacity
; ----------------------------------------------------------------
(Make-Package 'pd-load)
(Export '(states goal rules is-c-pred c-eq))
; ================================================================
(setq states '(
  ((at o1 A) (at o2 A) (atR A))
  ((insideR o1) (at o2 A) (atR A))
  ((insideR o1) (insideR o2) (atR A))
))

(setq goal '((insideR o1) (insideR o2)))

(setq rules
  '( (rule load
         (if (at (> x) A) (atR A))
         (then (load (< x)))
         (conq (add ((insideR (< x)))
                del ((at (< x) A))
         )     )
     )
))
; ---------------------------------------------------------------
; Predicates which indicate constructive rewriting
(defun is-c-pred (p-name)
  (or (equal p-name 'at) (equal p-name 'insideR))
)
; (at o1 x) and (at o2 x) => o1 = next(o2)
; (insideR o1) and (insideR o2) => o1 = next(o2)
(defun c-eq (p)
 (cond ((equal (nth 1 p) 'o1)
         (list (nth 1 p) '(next o2))  ; assoc-list (o1 (next o2))
        )
))
```



```
UNLOAD.PLAN
((AT O1 B) (AT O2 B) (ATR B))
(UNLOAD O1)
((INSIDER O1) (AT O2 B) (ATR B))
(UNLOAD O2)
((INSIDER O2) (INSIDER O1) (ATR B))
```

Figure 4.4: Output of DPlan for *unload*



```
LOAD.PLAN
((INSIDER O1) (INSIDER O2) (ATR A))
(LOAD O2)
((AT O2 A) (INSIDER O1) (ATR A))
(LOAD O1)
((AT O1 A) (AT O2 A) (ATR A))
```

Figure 4.5: Output of DPlan for *load*

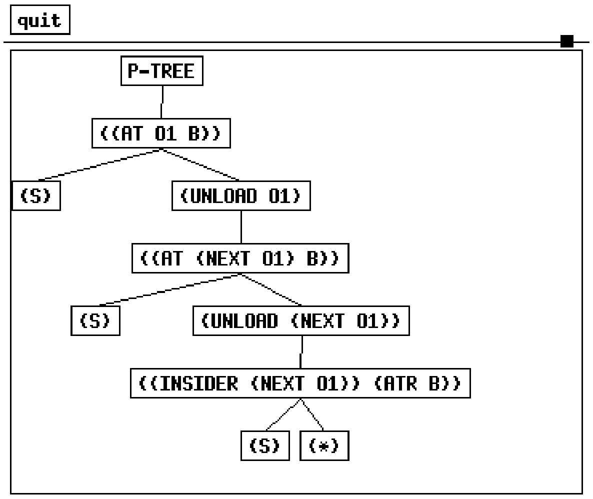


Figure 4.6: Binary tree for *unload*

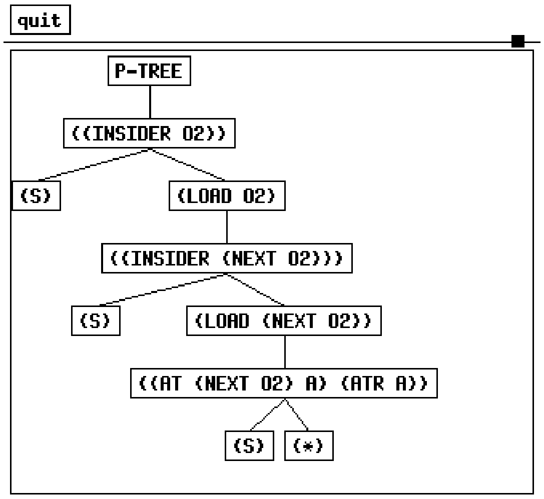


Figure 4.7: Binary tree for *load*

4.2 Complex cyclic macros: non-linear structures

Examples for non-linear structures are *tower* (presented in chapter 3.2 and discussed in [Wys87]), *rocket* and other transportation domains [VC93a], and *hanoi*. A non-linear D-plan not necessarily implies that the resulting cyclic macro has to be non-linear. The reason for this can be found in function theory [FH88]: A given syntactical realization of a function does not necessarily correspond with its semantic complexity [Hin78, Odi89]. For example, we can rewrite a tree-recursive function for calculating Fibonacci numbers into a linear recursive structure (by introducing an additional variable), or there exist loop-programs (corresponding to linear recursions) for *hanoi* problems [Er83, Pet84]. Similarly, a given sample of problem solutions – as represented in a D-plan – may have characteristics which make it possible to rewrite its given syntactical form into a computationally less complex structure.

We can think of three possible strategies to deal with non-linear DPlan outputs which we want to incorporate in our system:

- Using already acquired linear macros: We can provide our system first with simple linear problems and use the generalized linear macros when planning for more complex problems. For example, using the *unload* and *load* macros for solving the *rocket* problem results in a linear plan: *load objects, move rocket, unload objects.* This kind of "hierarchical" macro learning has two problems: (a) we need a teacher which provides the system with problems in such a sequence that hierarchical learning is possible; (b) during planning not only the operators given in the domain specification but additionally the macro memory has to be searched and applicability of macros has to be checked.
  Of course, the second problem has to be addressed anyway when we want to apply cyclic macros to make planning more efficient. Checking if a macro is applicable can be performed in two ways: either we index macros with semantical information (the predicates they fulfill and the domain in which they were learned) or we use pattern-matching between partial D-plan trees and unfolded cyclic macros (i.e. an initial program generated by expanding macro to a given depth)[1].

- Rewriting non-linear structures into linear structures if possible: That is the strategy we currently investigate. After constructively rewriting each linear path of the plan, we try to unify different planning paths. If the resulting structure is still a tree, we rewrite a "side-ordering" of paths (corresponding to a "case" statement) into a nested "sub-ordering" (corresponding to nested if-then-else statements) – that is, we convert a partial order into an arbitrary total order.

- Generating initial programs from non-linear plans: While linear macros

---

[1] Remark: Stefan Böhm is investigating this strategy in his diploma thesis.

are computationally more efficient than tree-recursive ones, they are not necessarily more efficient on the level of representation. Furthermore, a more compact (tree-) recursive macro is often easier to understand by humans than an optimized version because it is formulated in a "more natural" way (compare for example the tree and the linear recursive versions of *fibonacci* or *hanoi*).

- For the last two cases we may either generalize a "complex" macro, or try to learn encapsulated linear macros (as *load* and *unload*) in the context of a more complex problem (as *rocket*). The second strategy has to be based on a principle for segmenting a problem in independent subproblems (a syntactical approach for such segmentations is realized for example in ALPINE [Kno90]).

The rest of this section will be very superficial. Because transformation for non-linear plans is not fully implemented yet[2] we just present the original DPlan outputs and discuss some open problems.

4.2.1 Building a tower of alphabetically ordered blocks

Koza [Koz92] synthetisized a program for building a tower of blocks in a predefined order using genetic programming – that is, he provided the learning algorithm with a training set of initial states and the desired goal and inferred the *tower* function by search in the space of possible programs (see figure 4.8). This program does not guarantee that for each possible input the shortest operation sequence is executed: for initial states which correspond to towers sorted reverse to the goal tower it is not necessary to put all blocks on the table first. Instead, the tower can just reversed by exploiting the side-effect of *put* that a block gets cleared if the top of it is put on another block.

A program which builds a tower from arbitrary input states with optimal operation sequences is given in appendix B. The main function shows one possible realization of a cyclic macro for solving the tower problem:

```
(defun tower (l)
  (cond ((is-tower l) l)
        ((and (exists-tower l)
              (exist-free-neighbors l))
           (tower (put (scndgreatest l) (greatest l) l)))
        (T (tower (puttable (topof (greatest-no-tower l)) l)))
))
```

To generalize a macro corresponding to this program, it would be necessary to "invent" predicates (`is-tower`, `exists-tower`, `exists-free-neighbors`) and introduce more than one selector function (not only t̂opof as used in the *clearblock* macro, but also `greatest`, `scndgreatest`, and `greatest-no-tower`). Predicate invention is an established method in inductive logic programming

---

[2] Remark: I hope to be finished with a first prototype by Mai 1999.

program synthesized:
(EQ (DU (MT CS) (NOT CS)) (DU (MS NN) (NOT NN)))
a more readable version:
equal (
do move-to-table(x) until not(current-stack),
do move-to-stack(next-necessary-block) until not(next-necessary-block)
)

Genetic Programming:

- representing programs as function terms
- cross-over (syntactically correct) terms
- evaluate programs by their performance on a set of examples

Figure 4.8: Synthesizing *tower* with genetic programming

[FYar]. Therefore, it would be interesting to try to incorporate this idea in our plan transformation approach.

Another possible macro[3] is: apply *put(x, y)* for the desired goal order of blocks beginning from the base and apply *cleartop* to each block which is argument of *put*:

```
; l is the (reversed) goal sorting, f.e. (C B A)
; s is initialized with an initial state and modified by put and clearblock
; clearblock is a recursive function using puttable
(defun tower (l s)
  (cond ((null l) s)
        (T (tower (cdr l)
             (put (clearblock (cadr l) s) (clearblock (car l) s) s)))
)).
```

The strategy we currently investigate will generate non of these macros but a single complex function (see discussion of the *rocket* domain below).

4.2.2 Rocket problems for arbitrary numbers of objects

The DPlan domain specification for *rocket* is given in table 4.6.

There exists a second legal state fulfilling the goal – *((at o1 B) (at o2 B) (atR A))* – which we currently do not consider. The output of DPlan for the rocket domain is given in figure 4.9. If we allow different sequences to load or unload objects in the definition of states, the state ordering is not completely linear. There is always a branch dealing with an alternative ordering – but this branch is always pruned because the following states are already dealt with in another branch. This observation leads us to the following hypothesis for a

---

[3] proposed by Fritz Wysotzki 1998

Table 4.6: The rocket domain

```
; one-way rocket transportation problem, Veloso & Carbonell 93
; =================================================================
; two locations A and B and a oneway-st between
; transporter: R (rocket) can load objects (two objects o1, o2)
; -----------------------------------------------------------------
(Make-Package 'pd-rocket)
(Export '( states goal rules
           is-c-pred c-eq    ))
; =================================================================

(setq states '(
  ((at o1 B) (at o2 B) (atR B))
  ((insideR o1) (at o2 B) (atR B))   ; symmetric states
  ((insideR o2) (at o1 B) (atR B))   ; -''-
  ((insideR o1) (insideR o2) (atR B))
  ((at o1 A) (atR A) (at o2 B))
  ((insideR o1) (atR A) (at o2 B))
  ((at o2 A) (atR A) (at o1 B))
  ((insideR o2) (atR A) (at o1 B))
  ((at o1 A) (at o2 A) (atR A))
  ((insideR o1) (at o2 A) (atR A))   ; symmetric states
  ((insideR o2) (at o1 A) (atR A))   ; -''-
  ((insideR o1) (insideR o2) (atR A))  ))

(setq goal  '((at o1 B) (at o2 B)) )

(setq rules
  '( (rule move
         (if (atR A) )
         (then (moveR))
         (conq (add ((atR B))
                del ((atR A))  )     )  )
      (rule unload
          (if (insideR (> x)) (atR (> y)))
          (then (unload (< x)))
          (conq (add ((at (< x) (< y)))
                 del ((insideR (< x)))  )     )  )
     (rule load
         (if (at (> x) A) (atR A))
         (then (load (< x)))
         (conq (add ((insideR (< x)))
                del ((at (< x) A))  )     )  )      ))
; -------------------------------------------------------
; Predicates which indicate constructive rewriting
(defun is-c-pred (p-name)
  (or (equal p-name 'at) (equal p-name 'insideR))
)
; (at o1 x) and (at o2 x) => o2 = next(o1)
; (insideR o1) and (insideR o2) => o2 = next(o1)
(defun c-eq (p)
 (cond ((equal (nth 1 p) 'o2)
         (list (nth 1 p) '(next o1))  ; assoc-list (o2 (next o1))
        )
))
```

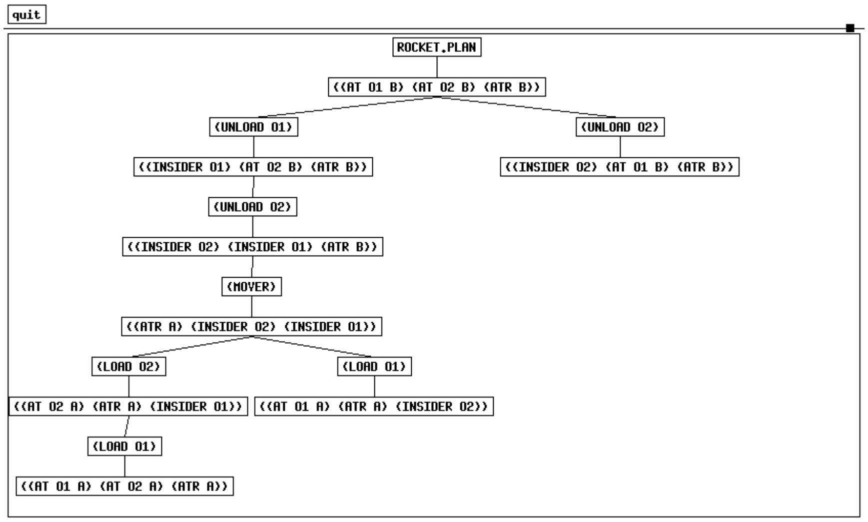


Figure 4.9: Output of DPlan for *rocket*

linearization heuristic: *If an operation leads to a leaf node and if this operation is also handled in another branch a a deeper level, then the operation and its succeeding leaf node can be deleted from the branch.*

If the branches for alternative orderings are deleted, all states are in an total order. Nevertheless, generalization is not so easy as for the problems discussed in section 4.1, because more than one operator is involved. The current implementation of our synthesis algorithm cannot deal with such inputs: it generates the hypothesis $g(at(x, B), B, unload(x, rocket(next(x), B)))$ which cannot be maintained when the *move* operator appears. To solve this problem, our synthesis algorithm has to be extended to subprogram construction. We are planning to extend our synthesis algorithm to subprogram generation – i.e. allowing to infer more than one recursive equation from an initial program[4].

Two programs solving the rocket domain for an arbitrary number of objects are given in appendix C. Both programs cover all possible input states – that is not only states where all objects are at location A at the beginning, but also problems, where some objects are already in the rocket or at the destination. The first program makes use of the linear macros for *unload* and *load*, the second program is a "complex" macro for solving the complete problem. For real world applications it is not acceptable, that we allow infinite capacity for the rocket. That is, we would need a second predicate for *load* which checks whether the rocket has still free capacity. This predicate has to be provided with the domain specification: the test $\not{f}ull(rocket)$ has to be included into the preconditions of *load*.

---

[4] Remark: This will probably done in the diploma thesis of Martin Mühlpfordt.

Table 4.7: The hanoi domain (static predicates are not listed for each state)

```
; Tower of Hanoi (3 discs)
; ==================================================================
(Make-Package 'pd-hanoi)
(Export '(states goal rules is-c-pred c-eq))
; ==================================================================
(setq states '(
  ((on d3 p1) (on d2 d3) (on d1 d2) (ct d1) (ct p2) (ct p3)  )
  ((on d3 p2) (on d2 d3) (on d1 d2) (ct d1) (ct p1) (ct p3)  )
  ((on d3 p3) (on d2 d3) (on d1 d2) (ct d1) (ct p1) (ct p2)  )
  ((on d3 p1) (on d2 d3) (on d1 p2) (ct d1) (ct d2) (ct p3)  )
  ((on d3 p1) (on d2 d3) (on d1 p3) (ct d1) (ct d2) (ct p2)  )
  ((on d3 p2) (on d2 d3) (on d1 p1) (ct d1) (ct d2) (ct p3)  )
  ((on d3 p2) (on d2 d3) (on d1 p3) (ct d1) (ct d2) (ct p1)  )
  ((on d3 p3) (on d2 d3) (on d1 p1) (ct d1) (ct d2) (ct p2)  )
  ((on d3 p3) (on d2 d3) (on d1 p2) (ct d1) (ct d2) (ct p1)  )
  ((on d2 p1) (on d1 d3) (on d3 p2) (ct d1) (ct d2) (ct p3)  )
  ((on d2 p1) (on d1 d3) (on d3 p3) (ct d1) (ct d2) (ct p2)  )
  ((on d2 p2) (on d1 d3) (on d3 p1) (ct d1) (ct d2) (ct p3)  )
  ((on d2 p2) (on d1 d3) (on d3 p3) (ct d1) (ct d2) (ct p1)  )
  ((on d2 p3) (on d1 d3) (on d3 p1) (ct d1) (ct d2) (ct p2)  )
  ((on d2 p3) (on d1 d3) (on d3 p2) (ct d1) (ct d2) (ct p1)  )
  ((on d3 p1) (on d1 d2) (on d2 p2) (ct d1) (ct d3) (ct p3)  )
  ((on d3 p1) (on d1 d2) (on d2 p3) (ct d1) (ct d3) (ct p2)  )
  ((on d3 p2) (on d1 d2) (on d2 p1) (ct d1) (ct d3) (ct p3)  )
  ((on d3 p2) (on d1 d2) (on d2 p3) (ct d1) (ct d3) (ct p1)  )
  ((on d3 p3) (on d1 d2) (on d2 p1) (ct d1) (ct d3) (ct p2)  )
  ((on d3 p3) (on d1 d2) (on d2 p2) (ct d1) (ct d3) (ct p1)  )
  ((on d3 p1) (on d1 p2) (on d2 p3) (ct d1) (ct d2) (ct d3)  )
  ((on d3 p1) (on d2 p2) (on d1 p3) (ct d1) (ct d2) (ct d3)  )
  ((on d1 p1) (on d3 p2) (on d2 p3) (ct d1) (ct d2) (ct d3)  )
  ((on d2 p1) (on d3 p2) (on d1 p3) (ct d1) (ct d2) (ct d3)  )
  ((on d1 p1) (on d2 p2) (on d3 p3) (ct d1) (ct d2) (ct d3)  )
  ((on d2 p1) (on d1 p2) (on d3 p3) (ct d1) (ct d2) (ct d3)  ) ))
; statics (currently included in every state)
;(smaller p1 d1) (smaller p1 d2) (smaller p1 d3) (smaller p2 d1)
;(smaller p2 d2) (smaller p2 d3) (smaller p3 d1) (smaller p3 d2)
;(smaller p3 d3) (smaller d2 d1) (smaller d3 d1) (smaller d3 d2)

(setq goal '((on d3 p3) (on d2 d3) (on d1 d2)))

(setq rules '(
    (rule move
        (if (on (> d) (> from)) (ct (< d)) (smaller (> to) (< d)) (ct (< to)))
        (then (move (< d) (< from) (< to)))
          (conq (add ((on (< d) (< to)) (ct (< from)))
                 del ((on (< d) (< from)) (ct (< to)))          )     ))))
```

4.2.3 Tower of Hanoi

The domain specification for *hanoi* is given in table 4.7. Here one shortcoming of our currently restricted specification language becomes obvious: static predicates (i.e. predicates which are never changed by an operator application – in contrast to fluents) cannot be defined globally for a domain but they have to be enumerated for each state. The graphical DPlan output is too large for one screen-shot. Therefore we will give a ‘ ‘hand-drawn” plan tree instead (see figure 4.10). A recursive program for solving Tower of Hanoi problems is given in appendix D.

4.3 Planning for list and number problems

Planning problems – as blocksworld, Tower of Hanoi, transportation, or scheduling problems – are typically not considered in program synthesis. On the other hand, number and list problems are only seldom considered in planning. Remember, that our planning approach is mainly intended for providing initial programs for program synthesis. To be able to compare our approach with

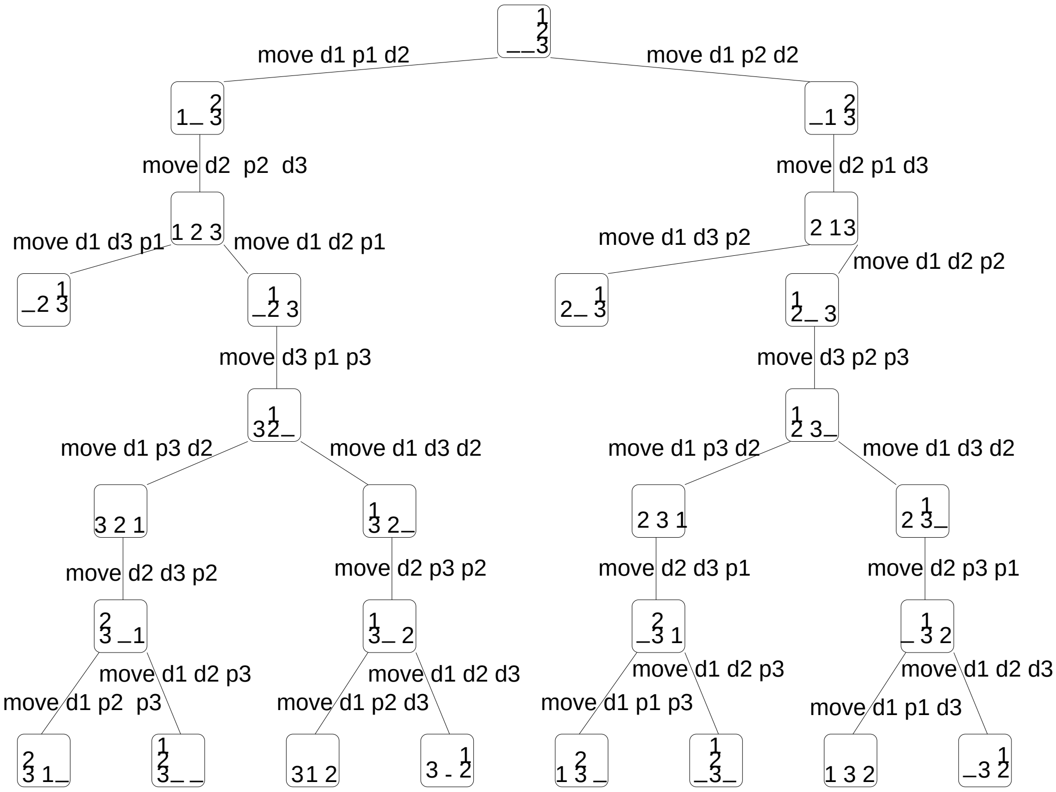


Figure 4.10: Output of DPlan for *hanoi* (abridged drawing)

other current (ILP) approaches (as FOIL [Qui90], GOLEM [MF90], or PROGOL [Mug95]), we have to be able to deal with the standard examples used in this area which are mostly list processing problems.

For a lot of typical planning problems there exist isomorphic or at least structural similar list and/or number problems [SW98]. For example, `append` has an nearly isomorphic structure to `loadob` and `unloadob` – the only difference is, that we need a selector (head) and a reductor (tail) function instead of only `next`:

$$append(l1, l2) = g(empty(l1), l2, cons(hd(l1), append(tl(l1), l2))).$$

The *clearblock* problem is similar to calculating the factorial of a number [SWar] – instead of the parameter s representing the (globally available) current situation we here have a constant output (1):

$$factorial(x) = g(eq0(x), 1, mult(x, factorial(pred(x)))).$$

While generalizing recursive marcos from initial programs for these kind of problems is easy [MS98], generating such initial programs with DPlan is currently not: Domain specifications for `append` and `factorial` cannot be represented in a natural way in the current version of DPlan because of our limited domain specification language[5]. In full ADL we could represent the *factorial* domain as:

mult(x,y):

PRECOND: $x \neq 0$

UPDATE: $x \leftarrow x \times (x - 1)$

fixed(x):

PRECOND: $x = 0$

UPDATE: $x \leftarrow 1$.

Allowing conditional effects, the operators could be integrated in a single one with no global precondition.

We give an example for modelling a list-domain without these features for *sort* (see table 4.8; see also a specification of this domain in PRODIGY in appendix E). This domain was originally introduced in [Wys87].

Again, we have to enumerate the static predicates for each state. The *swap* operator here is restricted to list neighbors – therefore, the intended cyclic macro is a *bubble-sort* algorithm. The DPlan output is given in figure 4.11.

---

[5] Remark: Extending our domain specification language will be worked on by Eckhard Wiederhold as his student project

Table 4.8: The sort domain

```
; sort
; ==================================================================
(Make-Package 'pd-sort)
(Export '(states goal rules is-c-pred c-eq))
; ==================================================================
(setq states '(
  ((isc p1 1) (isc p2 2) (isc p3 3) (db p1 p2) (db p2 p3))
  ((isc p1 1) (isc p2 3) (isc p3 2) (db p1 p2) (db p2 p3))
  ((isc p1 2) (isc p2 1) (isc p3 3) (db p1 p2) (db p2 p3))
  ((isc p1 2) (isc p2 3) (isc p3 1) (db p1 p2) (db p2 p3))
  ((isc p1 3) (isc p2 1) (isc p3 2) (db p1 p2) (db p2 p3))
  ((isc p1 3) (isc p2 2) (isc p3 1) (db p1 p2) (db p2 p3))
))

; isc = is-content position element
; db = directly before position_i position_j
; db is static

(setq goal '((isc p1 1) (isc p2 2) (isc p3 3)))

(setq rules
'( (rule swap
         (if (db (> p) (> q)) (isc (> p) (> n1)) (isc (> q) (> n2)))
         (then (swap (< p) (< q)))
         (conq (add ((isc (< p) (< n2)) (isc (< q) (< n1)))
                del ((isc (< p) (< n1)) (isc (< q) (< n2)))
         )      )
     )
))
```

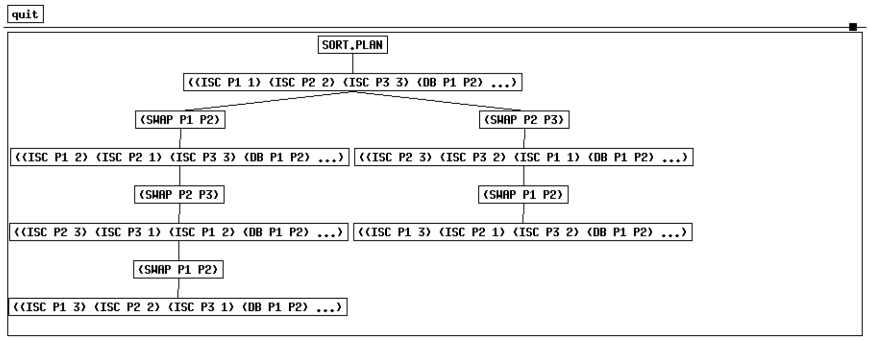


Figure 4.11: Output of DPlan for *sort*

Chapter 5

Conclusions and further work

We presented the state-based non-linear backward planner DPlan. DPlan is a universal planner for deterministic domains which is sound and complete and constructs optimal plans. The current implementation relies on sets of states as input, that is, planning is used to construct a complete partial order over states with respect to the number of operations needed to transform a state into a state fulfilling the planning goals. Providing such state sets results in an only polynomial effort for planning. Because DPlan is mainly intended as a tool for constructing initial programs as input into our program synthesis algorithm [SW98, MS98], we typically only plan for small domains (where it is easy to enumerate all states for an user). Our next implementation will be based on the alternative strategy described in chapter 3 – planning without initial states. Because we then have (1) to generate all possible predecessors for each state node, and (2) eliminate non-admissible and already considered states, planning effort will than be (as usual for generic planners) exponential in the worst case. The next version of DPlan will be able to use a more powerful ADL-like domain specification language. Our main concern is to introduce universal quantification and the possibility of applying functions.

Secondly, we presented our approach for transforming DPlan outputs into initial programs. The current implementation can only deal with linear plans, an extension to non-linear plans is under construction.

We believe that combining planning with program synthesis is fruitful for both areas of research: On the one hand, planning provides a more powerful tool for constructing initial (straight-forward) programs than the search and rewrite techniques usually employed in inductive program synthesis systems (see chapter 2). Planning is the natural approach to model the search of transformation sequences for inputs into desired outputs and it provides a standardized and powerful language for representing knowledge about primitive operators. We have extended this language to represent also knowledge about data-structures

– rewrite-rules to replace constants by constructive expressions. On the other hand, program synthesis provides a powerful approach to cyclic macro generalization. Our synthesis system [SW98] which is based on the theory of recursive functions and formalized in the framework of grammatical inference [MS98] provides a formally sound and purely syntactical approach to generalizing recursive functions from initial programs. The idea of cyclic macros can also be used to make universal planning more efficient: plans have only generated for small finite domains and then can be generalized to domains of arbitraty complexity. Finally, cyclic macros provide an alternative to learning control rules [VCP$^+$95]: a cyclic macro which in our approach is represented as recursive program scheme represents the complete subgoal structure of a domain. Therefore, if a cyclic macro is available, a plan can be generated completely without search by simply applying the macro to a given input state. To make cyclic macros available of planning, we have to provide a method for selecting an appropriate macro from memory. This is clearly a non-trivial problem; two ideas for macro retrieval are discussed at the beginning of section 4.2. A drawback in contrast to learning in PRODIGY [VCP$^+$95] is, that in our approach, learning cannot be performed incremental: To generalize a cyclic macro information about the transformation structure for the complete state set has to be available.

Clearly, there is still a lot to do, to bring our approach to a scope and a level of performance similar to PRODIGY [VCP$^+$95] – another system that combines planning and machine learning. But we strongly believe, that using the formal background of program synthesis for macro learning can open an new perspective on learning in planning.

Appendix A

Implementation of DPlan

```
README for DPlan

Ute Schmid Dec. 1998

available at http://ki.cs.tu-berlin.de/~schmid/DPlan-1.0/

-------------------------------------------------------------------------

DPlan consists of the following modules:

- dplan.lsp      : main module for planning and transformation of dplans
                   to initial programs
- plan-dstruc.lsp: global datastructure (each planning step as structure)
- ps-back.lsp    : backward application of sets of operators to current state
- c-rewrite.lsp  : rewriting of constants to constructive expressions
- p-xtree.lsp    : graphical output of trees via xtree (platform dependent!)
  - myxtree      : shell-script for calling xtree
  - xtree        : public domain program, see HELP-install

-------------------------------------------------------------------------

Currently implemented for Allegro Common Lisp
(previous version for clisp)

If want to avoid to adapt dplan to your platform
outcomment the following expressions in dplan.lsp

- loading of package p-xtree
- the calls of (xtree tree name) in function plan-control
- it is possible that you have to outcomment additionally
  (also in plan-control)
  * the external call of xmv ("save window with grafical output as give")
  * or adapt the syntax for calling external functions
    "user::run-shell-command ..."
    to your version of common lisp

-------------------------------------------------------------------------

Before starting dplan:
```

```
- make sure that you give the correct path for xtree in myxtree
  if you want to use the graphical output
- provide a problem specification in the correct format (required
  as input)
  -> see pd-cb, pd-tower, pd-rocket as examples

---------------------------------------------------------------------------

Running the planner:
call lisp, load dplan, call (start)
(input of a problem-specification file is handled interactively)

---------------------------------------------------------------------------
```

Appendix B

A recursive program for building towers

```
; Building a tower of alphabetically ordered blocks
; in blocksworlds with arbitrary numbers of blocks
; and for arbitrary initial states
; Ute Schmid Dec 4 97
; ----------------------------------------------------------------
; Representation: blocks names als numbers 1...n (instead of A, B, C...)
;                 each partial tower as list
; Input: list of lists
; Examples for the three blocks world
; ((1 2 3)) ((2 3) (1)) ((1) (2) (3)) ((2 1) (3)) ((3 2 1)) ((1 2) (3))
; ((1 3) (2)) ((3 1) (2)) ((3 2) (1)) ((1 3 2)) ((2 3 1)) ((2 1 3)) ((3 1 2))
; ----------------------------------------------------------------

; help functions
; ---------------

; flattens a list l
(defun flatten (l)
  (cond ((null l) nil)
        (T (append (car l) (flatten (cdr l))))
  )
)

; x+1 = y?
(defun onedif (x y)
  (= (1+ x) y)
)

; blocks world selectors
; ----------------------

; topmost block of a tower
(defun topof (tw)
 (car tw)
)

; bottom block (base) of a tower
(defun bottom (tw)
 (car (last tw))
)

; next tower
```

```
; f.e. ((2 1) (3))  -> (2 1)
(defun get-tower (l)
  (car l)
)

; tops of all current towers
(defun topelements (l)
  (sort (map 'list #'car l) #'>)
)

; topblock with highest number
(defun greatest (l)
  (car (topelements l))
)

; topblock mit second highest number
(defun scndgreatest (l)
 (cadr (topelements l))
)

; label of the block with the highest number
(defun maxblock (l)
  (cond ((null l) 0)
        (T (car (sort (flatten l) #'>)))
))

; find a tower (more than 1 block) which is not correctly sorted
(defun no-tower (l)
  (cond ((null l) nil)
        ((and (base (get-tower l) l) (sorted (get-tower l)))
                                                       (no-tower (cdr l)))
        ((single-block (get-tower l)) (no-tower (cdr l)))
        (T (car l))
))

; find all towers which are not correctly sorted
(defun get-all-no-towers (l)
 (cond ((null l) nil)
        ((and (base (get-tower l) l) (sorted (get-tower l)))
                                               (get-all-no-towers (cdr l)))
        ((single-block (get-tower l)) (get-all-no-towers (cdr l)))
        (T (cons (car l) (get-all-no-towers (cdr l))))
))


(defun find-greatest (max l)
  (cond ((null l) max)
        ((> (topof max) (topof (car l))) (find-greatest max (cdr l)))
        (T (find-greatest (car l) (cdr l)))
))

; find incorrect tower containing highest element
(defun greatest-no-tower (l)
 (cond ((null l) nil)
       (T (find-greatest (car (get-all-no-towers l))
                         (cdr (get-all-no-towers l))))
))


; blocksworld predicates
; ----------------------

; is tower only a single block?
(defun single-block (tw)
  (= (length tw) 1)
)

; has tower block with highest number as base?
```

```
(defun base (tw l)
  (equal (bottom tw) (maxblock l))
)

; exist two partial towers which top elements differ only by one?
(defun exist-free-neighbours (l)
  (onedif (scndgreatest l) (greatest l))
)

; exists a correct partial tower?
; f.e. (2 3) or (B C)
(defun exists-tower (l)
  (cond ((null l) nil)
        ((and (equal (bottom (get-tower l)) (maxblock l))
              (sorted (get-tower l))) T)
        (T (exists-tower (cdr l)))
))


; is block x predecessor to top of a tower?
(defun successor (x tw)
  (cond ((null tw) T)
        ((onedif x (car tw)) T) ;(successor x (cdr tw)))
        (T nil)
))

; is tower sorted?
(defun sorted (tw)
 (cond ((null tw) T)
       ((successor (car tw) (cdr tw)) (sorted (cdr tw)))
       (T nil)
))

; exists only one tower?
(defun single-tower (l)
  (null (cdr l))
)

; goal state?
(defun is-tower (l)
 (and (single-tower l) (sorted (get-tower l)))
)

; -------------------------------------------------------------------

; blocksworld operators
; ---------------------

; put x on y
(defun put (x y l)
  (cond ((null l)
                  (print 'put) (print x) (print y)
                  nil)
        ((equal (caar l) x) (cond ((not (null (cdar l)))
                                  (append (list (cdar l)) (put x y (cdr l))))
                                  (T (put x y (cdr l)))))
        ((equal (caar l) y) (cons (cons x (car l)) (put x y (cdr l))))
        (T (cons (car l) (put x y (cdr l))))
))

; puttable x
(defun puttable (x l)
  (cond ((null l) nil)
        ((equal (caar l) x) (print 'puttable) (print x)
                            (cons (list x) (cons (cdar l) (cdr l))))
        (T (cons (car l) (puttable x (cdr l))))
))
```

```
; ------------------------------------------------------------------

; main function
; -------------

(defun tower (l)
  (cond ((is-tower l) l)
        ((and (exists-tower l)
              (exist-free-neighbours l))
           (tower (put (scndgreatest l) (greatest l) l)))
        (T (tower (puttable (topof (greatest-no-tower l)) l)))
))
```

Appendix C

Recursive programs for the rocket domain

```
;; Iterative Macro for the Rocket Domain
;; (assuming a rocket with infinete capacity)
;; Ute Schmid 11/23/98
;; ----------------------------------------------------------
;;
;; call (rocket boxes-at-A boxes-at-B boxes-in-Rocket position-of-Rocket)
;;      cf. (rocket '(b1 b2 b3) nil nil 'A)
;;          (rocket '(b1) '(b2 b3) 'A))
;; etc.
;;
;; ---------------------------------------------------------
;;
;; 1st variant: recursive loadb and unload
;; problem: call of rocket after loadb relies on instantiation of
;; boxes-at-A with nil
;;
;; alternatives: global variables
;;               loadb returns a list of boxes-at-A and boxes-in-Rocket
;;                 without let loadb would be evaluated twice!!
;;
;; ---------------------------------------------------------


;; --- primitive opns --- ;;

(defun empty (l)
  (null l)
)

(defun put (e l)
  (cons e l)
)

(defun next-box (l)
  (car l)
)

(defun rest-boxes (l)
  (cdr l)
)

;; --- drive --- ;;
```

```
(defun drive-to-B (position)
  (cond ((eq position ’A) (print ’rocket-arrived-at-B) ’B)
        ( T nil)
))


;; --- recursive load and unload --- ;;

(defun unload (position boxes-in-Rocket boxes-at-B)
(print ‘(...unloading box ,@(next-box boxes-in-Rocket)))
  (cond ((empty boxes-in-Rocket) boxes-at-B)
        (T (put (next-box boxes-in-Rocket)
                (unload position (rest-boxes boxes-in-Rocket) boxes-at-B)
        )  )
))


(defun loadb (boxes-at-A boxes-in-Rocket)
(print ‘(...loading box ,@(next-box boxes-at-A)))
  (cond ((empty boxes-at-A) boxes-in-Rocket)
        (T (put (next-box boxes-at-A)
                (loadb (rest-boxes boxes-at-A) boxes-in-Rocket)
        )  )
))

;; --- ROCKET --- ;;

(defun rocket (boxes-at-A boxes-at-B boxes-in-Rocket position-of-Rocket)
(print ‘(boxes-at-A ,@boxes-at-A boxes-at-B ,@boxes-at-B
         boxes-in-Rocket ,@boxes-in-Rocket
         position-of-Rocket ,@position-of-Rocket))
  (cond ((empty boxes-at-A)
         (cond ((empty boxes-in-Rocket)
                (print ’done)
                T
               )
               ((eq ’A position-of-Rocket)
                (print ’driving-to-dest-and-unloading)
                (unload (drive-to-B position-of-Rocket) boxes-in-Rocket
                         boxes-at-B)
               )
               (T (print ’cannot-reach-goal) nil)
        ))
        (T
         (cond ((eq ’B position-of-Rocket)
                (print ’cannot-reach-goal)
                nil
               )
               (T
                (print ’loading-boxes)
                (rocket nil boxes-at-B
                        (loadb boxes-at-A boxes-in-Rocket)
                        position-of-Rocket
               ))
        ))
))

;; Iterative Macro for the Rocket Domain
;; (assuming a rocket with infinete capacity)
;; Ute Schmid 11/23/98
;; -----------------------------------------------------------
;;
;; call (rocket boxes-at-A boxes-at-B boxes-in-Rocket position-of-Rocket)
;;      cf. (rocket ’(b1 b2 b3) nil nil ’A)
;;          (rocket ’(b1) ’(b2 b3) ’A))
;; etc.
;;
;; ----------------------------------------------------------
```

```
;;
;; 2nd variant: recursive rocket only
;;
;; ---------------------------------------------------------


;; --- primitive opns --- ;;

(defun empty (l)
  (null l)
)

(defun put (e l)
  (cons e l)
)

(defun next-box (l)
  (car l)
)

(defun rest-boxes (l)
  (cdr l)
)

;; --- drive --- ;;

(defun drive-to-B (position)
  (cond ((eq position 'A) (print 'rocket-arrived-at-B) 'B)
        ( T nil)
))


;; --- ROCKET --- ;;

(defun rocket (boxes-at-A boxes-at-B boxes-in-Rocket position-of-Rocket)
(print `(boxes-at-A ,@boxes-at-A boxes-at-B ,@boxes-at-B
         boxes-in-Rocket ,@boxes-in-Rocket
         position-of-Rocket ,@position-of-Rocket))
  (cond ((empty boxes-at-A)
         (cond ((empty boxes-in-Rocket)
                (print 'done)
                T
               )
               ((eq 'A position-of-Rocket)
                (print 'driving-to-dest)
                (rocket boxes-at-A boxes-at-B boxes-in-Rocket
                        (drive-to-B position-of-Rocket))
               )
               ((eq 'B position-of-Rocket)
                (print 'unloading-boxes)
                (rocket boxes-at-A (put (next-box boxes-in-Rocket) boxes-at-B)
                        (rest-boxes boxes-in-Rocket) position-of-Rocket)
               )
               (T (print 'cannot-reach-goal) nil)
        ))
        (T
         (cond ((eq 'B position-of-Rocket)
                (print 'cannot-reach-goal)
                nil
               )
               (T
                (print 'loading-boxes)
                (rocket (rest-boxes boxes-at-A) boxes-at-B
                        (put (next-box boxes-at-A) boxes-in-Rocket)
                        position-of-Rocket)
               )
        ))
))
```

Appendix D

A recursive program for Tower of Hanoi with arbitrary input states

```
; Towers of Hanoi generalized for arbitrary initial states
; Ute Schmid
; =============================================================
; (1) Load
; (2)                        C = goal peg
;     (SETQ A ’(1 2 3) B NIL C NIL)    or
;     (SETQ A ’(1 2)   B NIL C ’(3))   or
;     (SETQ A ’(1)     B NIL C ’(1 2)) or
;     (SETQ A ’(2 3)   B NIL C ’(1))   or
;     etc.                               for 3 discs: 27 poss.
; (3) (ghanoi)

; ==============================================================

; help functions

; on: returns peg on which a disc lies

(DEFUN ison (disc peg)
  (COND ( (NULL peg) NIL)
        ( (= (CAR peg) disc) T)
        ( T  (ison disc (cdr peg)))
  )
)

(DEFUN on (disc current-peg goal-peg inter-peg)
 (COND ( (ison disc (eval current-peg))  current-peg)
       ( (ison disc (eval goal-peg)) goal-peg)
       ( T                           inter-peg)
 )
)

; interpeg: returns peg which is neither the peg on which the currently
; regarded disc lies nor the current goal-peg

(DEFUN interpeg (disc peg)
 (COND ( (or (and (equal (on disc ’A ’B ’C) ’B) (equal peg ’C))
             (and (equal (on disc ’A ’B ’C) ’C) (equal peg ’B))
         ) ’A)
```

```
       ( (or (and (equal (on disc ’A ’B ’C) ’A) (equal peg ’C))
             (and (equal (on disc ’A ’B ’C) ’C) (equal peg ’A))
        ) ’B)
       ( (or (and (equal (on disc ’A ’B ’C) ’A) (equal peg ’B))
             (and (equal (on disc ’A ’B ’C) ’B) (equal peg ’A))
         ) ’C)
 )
)

; topof: returns top disc of a peg

(DEFUN topof (peg)
 (COND ( (null (car (eval peg))) nil)
       ( T   (car  (eval peg)))
 )
)

; clearpeg: true, if no disc is on peg. false otherwise

(DEFUN clearpeg (peg)
 (COND ( (null (car (eval peg))) T)
       ( T nil)
 )
)

; cleartop: true if no disc is on top of the current disc, false otherwise

(DEFUN cleartop (disc)
 (COND ( (and (equal (on disc ’A ’B ’C) ’A) (= (car A) disc))
         T)
       ( (and (equal (on disc ’A ’B ’C) ’B) (= (car B) disc))
         T)
       ( (and (equal (on disc ’A ’B ’C) ’C) (= (car C) disc))
         T)
       ( T nil)
 )
)


; movedisc: put disc to peg

(DEFUN movedisc (disc peg)
  (COND ( (= disc 0) (PRINT (LIST ’NO ’DISC)))
        ( (equal (on disc ’A ’B ’C) peg)
          (PRINT (LIST ’Disc disc ’IS ’ON ’PEG peg))
        )
        ( (OR (clearpeg peg) (> (topof peg) disc))
          (PRINT
             (LIST ’MOVE ’DISC disc
                   ’FROM       (on disc ’A ’B ’C)
                   ’TO         peg
          )  )
          (SET (on disc ’A ’B ’C) (CDR (eval (on disc ’A ’B ’C))))
          (SET peg (CONS disc (EVAL peg)))
        )
        ( T  (PRINT (LIST ’MOVING disc ’TO peg ’IMPOSSIBLE )))
  )
)


; move: the recursive function for calculating the goal-stack for
; solving tower of hanoi

(DEFUN move (disc peg)
 (COND ( (and (= disc 1) (equal (on disc ’A ’B ’C) peg)) T )

 ( T (COND
       ( (equal (on disc ’A ’B ’C) peg)
```

```
         (move (- disc 1) peg)
       )
       ( (and (not (equal (on disc ’A ’B ’C) peg))
              (not (and (cleartop disc) (clearpeg peg)))
              (> disc 1))
         (move (- disc 1) (interpeg disc peg))
       )
     )
   (movedisc disc peg)
   (COND ((> disc 1) (move (- disc 1) peg)))
 )
 )
)

; ................................................................

(DEFUN number-of-discs (p1 p2 p3)
   (+ (LENGTH p1) (+ (LENGTH p2) (LENGTH p3)))
)

; move: number of discs x goal-peg -> solution

(DEFUN ghanoi ()
  (move (number-of-discs A B C) ’C)
)
```

Appendix E

List Sorting in PRODIGY

Prodigy offers the possibility to use infinite types and external functions. I wanted to define an inifinite type *number* and an external function which returns the content of a list-position. But, when I tried to specify the list sorting problem with these features, it did not work[1]. Therefore, I modelled the list-sorting domain in a standard way: using*Some-Number* instead of the built-in lisp-predicate *numberp* and explicitely coding the *is-content* predicate.

---

[1] A meeting with Eugene Fink, Nov. 1998, brought the result, that this kind of specification is currently not possible. I will check again with Manuela Veloso.

```
;;; Domain:  Lists (flat, over nat-numbers)
(create-problem-space 'lists :current t)
(ptype-of Position :top-type)
(ptype-of Content :top-type) ; represents the current content of a position
(defun Some-Number () '(1 2 3 4 5 6))
; ----------------
(operator SWAP
  (params <p1> <p2>)
  (preconds
    ((<p1> POSITION)
     (<p2> POSITION)
     (<n1> (and CONTENT (Some-Number)))
     (<n2> (and CONTENT (Some-Number)))
    )
    (and (directly-before <p1> <p2>)
         (is-content <p1> <n1>)
         (is-content <p2> <n2>))
  )
  (effects
    ()
    ((del (is-content <p1> <n1>))
     (del (is-content <p2> <n2>))
     (add (is-content <p1> <n2>))
     (add (is-content <p2> <n1>))
    )
  )
)
```

Figure E.1: Definition of the `list`-domain in PRODIGY

```
;;; example problem
;;; ---------------
;;; sort list of 3 elements; instantiate with
;;; [1 2 3] 0 swaps
;;; [1 3 2] swap(3,2)
;;; [2 1 3] swap(2,1)
;;; [2 3 1] swap(3,1),swap(2,1)
;;; [3 1 2] swap(3,1),swap(3,2)
;;; [3 2 1] swap(3,2),swap(3,1),swap(2,1)
(setf (current-problem)
      (create-problem
       (name sort1)
       (objects
        (pos1 pos2 pos3 position)
       )
       (state
        (and (directly-before pos1 pos2)  ; static
             (directly-before pos2 pos3)  ; static
             (is-content pos1 3)
             (is-content pos2 2)
             (is-content pos3 1)
       ))
       (goal
        (and (is-content pos1 1)
             (is-content pos2 2)
             (is-content pos3 3)
        ))
)    )
```

Figure E.2: Specification of a list-sorting problem in PRODIGY

Bibliography

[AHE90] J. Allen, J. Hendler, and A. Tate (Eds.). *Readings in Planning.* Morgan Kaufmann, San Mateo, CA, 1990.

[And86] J.R. Anderson. Knowledge compilation: A general learning mechanism. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning - An Artificial Intelligence Approach*, volume 2, pages 289–310. Tioga, 1986.

[BCE$^+$92] Jim Blythe, Jaime G. Carbonell, Oren Etzioni, Yolanda Gil, Robert Joseph, Dan Kahn, Craig Knoblock, Steven Minton, Alicia Perez, Scott Reilly, Manuela Veloso, and Xuemei Wang. PRODIGY 4.0: The manual and tutorial. Technical Report CS-92-150, Carnegie Mellon University, School of Computer Science, June 1992.

[BF97] A.L. Blum and M. Frust. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[BV96] Daniel Borrajo and Manuela Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning*, 10:1–34, 1996.

[BV97] J. Blythe and M. Veloso. Analogical replay for efficient conditional planning. In *AAAI-97*, 1997.

[BW94] A. Barrett and D. Weld. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67:71–112, 1994.

[CC86] P. Cheng and J. Carbonell. The FERMI system: Inducing iterative macro-operators from experience. In T. Kehler and S. Rosenschein, editors, *Proceedings of the 5th National Conference on Artificial Intelligence*, volume 1, pages 490–495, Los Altos, CA, USA, August 1986. Morgan Kaufmann.

[Cha87] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[CN78] 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.

[Coh98] W.W. Cohen. Hardness results for learning first-order representations and programming by demonstration. *Machine Learning*, 30:57–97, 1998.

[CRT98] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. Technical Report 9801-10, IRST, Trento, Italy, 1998.

[DC96] Robert Driskill and Jaime Carbonell. Search control in problem solving: A gapped marco operator approach. unpublished research report, 1996.

[DGR98] M. Di Manzo, E. Giunchiglia, and S. Ruffino. Planning via model checking in deterministic domains: Preliminary report. *Lecture Notes in Computer Science*, 1480:221–??, 1998.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1(5):269–271, 1959.

[ea94] K. Erol et al. Htn planning: Complexity and expressivity. In *Proceedings of AAAI 1994, Seattle WA*, 1994.

[Er83] M. C. Er. An iterative solution to the generalized towers of hanoi problem. *BIT*, 23:295–302, May 1983.

[FH88] A.J. Field and P.G. Harrison. *Functional Progamming.* Addison-Wesley, Reading, MA, 1988.

[FN71a] R. E. Fikes and H. J. Nilsson. STRIP: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:pages 189–205, 1971.

[FN71b] R.E. Fikes and N.J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1971.

[FY95] Eugene Fink and Qiang Yang. Planning with primary effects: Experiments and analysis. In *IJCAI-95*, pages 1606–1611, 1995.

[FYar] P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. submitted paper, to appear.

[GH85] R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):143–57, 1985.

[Gre69] C. Green. Application of theorem proving to problem solving. Technical report, IJCAI 1, 1969.

[Hin78] P.G. Hinman. *Recursion-Theoretic Hierarchies.* Springer, New York, 1978.

[Kla78] D. Klahr. Goal formation, planning, and learning by preschool problem solvers or: "my socks are in the dryer". In *Children's thinking: What develops?* Erlbaum, Hillsdale, NJ, 1978.

[KNH97] J. Koehler, B. Nebel, and J. Hoffmann. Extending planning graphs to an ADL subset. In *ECP-97 and extended version as Technical Report No. 88/1997, University Freiburg*, 1997.

[Kno90] Craig Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of AAAI 1990*, 1990.

[Kor85] Richard E. Korf. Macro-operators: a weak method for learning. *Artificial Intelligence, 1985*, 26:35–77, 1985.

[Koz92] J. Koza. *Genetic programming: On the programming of computers by means of natural selection.* MIT Press, Cambridge, MA, 1992.

[Koz94] John R. Koza. Introduction to genetic programming. In Kenneth E. Kinnear, editor, *Advances in Genetic Programming*, Complex Adaptive Systems, pages 21–42, Cambridge, 1994. MIT Press.

[KS98] H. Kautz and B. Selman. Blackbox: A new approach to the application of theorem proving to problem solving. In *AIPS98 Workshop on Planning and Combinatorial Search*, pages 58–60, 1998.

[LD94] N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, London, 1994.

[McC98] J. McCarthy. Programs with common sense. In M. Minsky, editor, *Semantic Information Processing*, pages 403–418. MIT Press, Cambridge, MA, 1998.

[MDR94] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming, Special Issue on 10 Years of Logic Programming*, 19-20:629–679, 1994.

[MF90] S. Muggleton and C. Feng. Efficient induction of logic programs. *Proc. of the Workshop on Algorithmic Learning Theory by the Japanese Society for AI*, pages 368–381, 1990.

[Min85] Steven Minton. Selectively generalizing plans for problem-solving. In *Proceedings of the IJCAI-85*, pages 596–599, 1985.

[Mit97] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[MKKC86] T.M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986.

[MR91] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *AAAI-91*, volume 2, pages 634–639, 1991.

[MS98] M. Mühlpfordt and U. Schmid. Synthesis of recursive functions with interdependent parameters. In *Proc. of the Annual Meeting of the GI Machine Learning Group, FGML-98*, pages 132–139, TU Berlin, 1998.

[Mug95] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[MW87] Z. Manna and R. Waldinger. How to clear a block: a theory of plans. *Journal of Automated Reasoning*, 3(4):343–378, December 1987.

[NR81] A. Newell and P.S. Rosenbloom. Mechanisms of skill acquisition and the law of practice. In J.R. Anderson, editor, *Cognitive skills and their acquisition.* Erlbaum, Hillsdale, N.J., 1981.

[NS61] A. Newell and H.A. Simon. Gps, a program that simulates human thought. In H. Billing, editor, *Lernende Automaten*, pages 109–124. Oldenbourg, München, 1961.

[Odi89] P. Odifreddi. *Classical Recursion Theory*, volume 125 of *Studies in Logic.* North-Holland, Amsterdam, 1989.

[Ped89] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. of the First Knowledge Representation Conference*, pages 324–332, San Mateo, CA, 1989. Morgan Kaufmann.

[Pet84] A. Pettorossi. Towers of hanoi problems: Deriving the iterative solutions using the program transformation technique. Technical Report R.82, Instituto di Analisi dei Sistemi ed Informatica, Roma, March 1984. ki-general, inf-ap.

[PS92] M. Peot and D. Smith. Conditional nonlinear planning. In *1st International Conference on AI Planning Systems, AIPS-92*, pages 189–197. Morgan Kaufmann, 1992.

[PW92] J. S. Penberthy and D. S Weld. UCPOP: A sound, complete partial order planner for adl. In *Proc. 3rd Int. Conf. on Principles in Knowledge Representation and Reasoning*, pages 103–114, 1992.

[Qui90] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[RN86] P. S. Rosenbloom and A. Newell. The chunking of goal hierarchies: A generalized model of practice. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning - An Artificial Intelligence Approach*, volume 2, pages 247–288. Morgan Kaufmann, 1986.

[RN95] S. J. Russell and P. Norvig. *Artificial Intelligence. A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Sac74] E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Sac75] E.D. Sacerdoti. The nonlinear nature of plans. *Noteford Rev. Inst Technical Note*, 101. Inst Technical Noteford Rev. Inst Technical Note:101, 1975.

[SC89] P. Shell and J. Carbonell. Towards a general framework for composing disjunctive and iterative macro-operators. In *11th IJCAI-89*, Detroit, MI, 1989.

[Sch87] M.J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *IJCAI '87*, pages 1039–1046, 1987.

[Sch95] M. Schoppers. The use of dynamics in an intelligent controller for a space faring rescue robot. *Artificial Intelligence*, 73(1-2):175–230, 1995.

[Sum77] P. D. Summers. A methodology for LISP program construction from examples. *Journal ACM*, 24(1):162–175, 1977.

[Sus75] G.J. Sussman. *A Computer Model of Skill Acquisition*. American Elsevier, New York, 1975.

[SW98] U. Schmid and F. Wysotzki. Induction of recursive program schemes. In *Proceedings of the 10th European Conference on Machine Learning (ECML-98)*, number 1398 in LNAI, pages 228–240. Springer, 1998.

[SWar] U. Schmid and F. Wysotzki. Skill acquisition can be regarded as program synthesis: An integrative approach to learning by doing and learning by analogy. In *Mind Modelling - A Cognitive Science Approach to Reasoning, Learning and Discovery*. Pabst Science Publishers, Lengerich, to appear.

[Tar83] R. E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, 1983.

[Tat75] A. Tate. Interacting goals and their use. In *IJCAI-75*, pages 215–218, 1975.

[VC93a] M. M. Veloso and J. G. Carbonell. Towards scaling up machine learning: A case study with derivational analogy in prodigy. In S. Minton, editor, *Machine Learning Methods for Planning*, chapter 8. Morgan Kaufmann, 1993.

[VC93b] Manuela M. Veloso and Jaime G. Carbonell. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning*, 10(3):249–278, March 1993.

[VCP+95] M. Veloso, J. Carbonell, M. A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The prodigy architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[Vel94] M. Veloso. *Planning and Learning by Analogical Reasoning.* Springer, 1994.

[Wal75] R. Waldinger. Achieving several goals simultanously. In *Machine Intelligence*, volume 8, pages 94–138. Ellis Horwood, Chichester, England, 1975.

[Wel94] D. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[Wel99a] D. Weld. Programming by demonstration: An inductive learning formulation. In *IUI-99*, 1999.

[Wel99b] D. Weld. Recent advances in ai planning. *(to appear in AI Magazine, 1999) (AAAI-98, July 1998)*, 1999.

[WH89] P.H. Winston and B.K.P. Horn. *LISP.* Addison-Wesley, Reading, Mass, 1989.

[Win92] P.H. Winston. *Artificial Intelligence, 3rd Edition.* Addison-Wesley, Reading, MA, 1992.

[Wys83] F. Wysotzki. Representation and induction of infinite concepts and recursive action sequences. In *Proceedings of the 8th IJCAI*, Karlsruhe, 1983.

[Wys87] 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.