Applying Inductive Program Synthesis to Learning Domain-Dependent Control Knowledge — Transforming Plans into Programs

Ute Schmid Fritz Wysotzki[1]

June 2000

CMU-CS-00-143

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This report was written while the first author was visiting researcher at CMU, supported by a DFG research scholarship, invited by Jaime Carbonell. The contact address for both authors is: *Dept. of Computer Science, Technical University Berlin, Franklinstr. 28, D-10587 Berlin, Germany*. For obtaining the programs described here, write an email to `schmid@cs.tu-berlin.de` or see `http://ki.cs.tu-berlin.de/~schmid`.
The report gives an extended and updated presentation of the work reported at AIPS-00 (Schmid & Wysotzki, 2000).

[1]Dept. of Computer Science, Technical University Berlin, Franklinstr. 28, D-10587 Berlin, Germany

We want to thank Jaime Carbonell and Rune Jensen for helpful discussions, and Peter Geibel, Martin Mühlpfordt, Heike Pisch, and Bernhard Wolf for helpful comments on different parts of this paper.

Keywords: control knowledge learning, universal planning, inductive program synthesis, data type inference

Abstract

The goal of this paper is to demonstrate that inductive program synthesis can be applied to learning domain-dependent control knowledge from planning experience. We represent control rules as recursive program schemes (RPSs). An RPS represents the complete subgoal structure of a given problem domain with arbitrary complexity (e. g., *rocket* transportation problem with n objects). That is, if an RPS is provided for a planning domain, search can be omitted by exploiting knowledge of the domain. We propose the following steps for automatical inference of control knowledge: (1) Exploring a problem with small complexity (e. g., *rocket* with 3 objects) using an universal planning technique, (2) transforming the universal plan into a finite program, and (3) generalizing this program into an RPS. While generalization can be performed purely syntactical, plan transformation is knowledge dependent. Our approach to folding finite programs into RPSs is reported in detail elsewhere. In this report we focus on plan transformation. We propose that inferring the data type underlying a given plan provides a suitable guideline for plan-to-program transformation.

Contents

Chapter 1

Introduction

During the last years, a number of efficient domain-independent planning algorithms have been proposed (e. g., Blum & Furst, 1997; Koehler, Nebel, & Hoffmann, 1997; Kautz & Selman, 1998; Long & Fox, 1999; Bonet & Geffner, 1999). Nevertheless, the problem of scaling-up such search-based algorithms to complex real-world problems (e. g., the logistics domain with many objects to transport from and to a large set of different locations by means of different vehicles[1]) remains – due to the inherent complexity of the planning task. The obvious remedy is to guide planning by domain specific control knowledge. For example, in the *rocket* one-way transportation domain (Veloso & Carbonell, 1993) search for a solution can be speed-up considerably, if the knowledge that all objects have to be loaded into the rocket before it flies to its destination is provided. On the other hand, providing planning systems with domain specific knowledge would require to put more effort in pre-planning analysis and specification of a domain which is in conflict with the idea of general-purpose planning systems. Consequently, current research focusses on the automatic inference of domain specific characteristics by static analysis of the current domain (Long & Fox, 2000) and on learning control rules from planning (Martín & Geffner, 2000).

In our work, we focus on *learning* domain specific control knowledge from some initial planning experience. There are different approaches addressing control knowledge learning reported in literature: The earliest approaches are concerned with learning linear macro-operators (Minton, 1985; Korf, 1985). Interest in this approach has decreased over the last decade – mainly because of the utility problem (Minton, 1985). But new results in rein-

[1]see AIPS98 and AIPS00 planning competitions, `http://www.cs.toronto.edu/aips2000/`

forcement learning are promising – showing that more complex problems are solvable and that planning can be speed-up considerably when applying macros (Precup & Sutton, 1998). Learning linear macros, similar to other incremental approaches to control knowledge learning (Borrajo & Veloso, 1996), aim at *reducing* the planning effort by providing guidelines to the search engine, especially with respect to goal-ordering. In contrast, the goal of our approach is to *eliminate* search completely.

An intuitive way to represent control knowledge of a domain are (functional) programs: A recursive program[2] represents the complete solution strategy (i. e., subgoal-ordering) of a domain. Therefore, we propose to apply a technique of *inductive program synthesis* (Summers, 1977; Wysotzki, 1983; Schmid & Wysotzki, 1998) to control knowledge learning. Inductive program synthesis algorithms learn recursive programs from a small set of input/output examples. Learning is performed by a two-step process: in a first step, I/O examples are transformed into a finite program; in a second step, the finite program is generalized to a recursive program. This second step is also called generalization-to-n and corresponds to programming by demonstration (Cohen, 1998). While the mutual benefit of combining planning and program synthesis is recognized in the deductive field (Manna & Waldinger, 1987), there is only limited cross-fertilization of planning and *inductive* program synthesis or other machine learning approaches: Shavlik (1990) demonstrates how explanation-based learning (inductive logic programming) can be applied to learning recursive concepts from problem solving examples; Shell and Carbonell (1989) show analytically and empirically how iterative macros can reduce planning effort and point out that learning iterative macros has to rely on generalization-to-n algorithms; Kalmar and Szepesvari (1999) discuss learning and efficiency of iterative macros in the context of Markov decision problems; (Koza, 1992) applies genetic programming to learning a functional program for solving the *tower* problem.

A different learning approach is proposed by Martín and Geffner (2000): Instead of recursive (or iterative) programs, decision lists representing generalized policies are induced from example state-action pairs (see also Briesemeister, Scheffer, & Wysotzki, 1996). While a recursive program generates the complete transformation sequence for any given input state into a goal state (e. .g., by means of an eval-apply-interpreter), the inferred rules are applied "step-by-step": for a current state the appropriate rule is selected

[2]More precisely, in our work we synthesize sets of recursive functions. Synthesis is not restricted to a given programming language – program terms are considered as elements of some term algebra, instead. Therefore, we infer *recursive program schemes* (RPSs) and if we talk of functions or programs we refer to RPSs.

and applied, the resulting successor state is again checked against the rules and so on.

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

In this paper, we focus on planning and plan transformation. Our work covering the second step of program synthesis – generalization-to-n –, together with arguments in what way program synthesis can profit from incorporating planning techniques, is reported in detail elsewhere (Wysotzki, 1983; Schmid & Wysotzki, 1998; Schmid, Mühlpfordt, & Wysotzki, 1999). In the next chapter we give an overview over the principal components of our approach – universal planning, plan transformation, and generalization-to-n – using the *rocket* domain for illustration. In chapter 3 we introduce our universal planning system, in chapter 4 we present plan-transformation and the resulting recursive programs for a variety of domains, and in chapter 5 we conclude with an evaluation of our approach and further work to be done.

Chapter 2

Learning Control Rules from Plans

2.1 Optimal Universal Plans

Our planning system DPlan is designed as a tool to support the first step of inductive program synthesis – generating finite program traces for transforming input examples into the desired output. Because our work is in the context of program synthesis, planning is for deterministic and (small) finite domains[1] only and completeness and optimality are of more concern than efficiency considerations. DPlan is a state-based, non-linear, total-order backward planner. Our algorithm is named DPlan in reference to the Dijkstra-algorithm, because it is single-source-shortest-paths algorithm with the states fulfilling the top-level goals as source. DPlan is similar to universal planning (Schoppers, 1987; Cimatti, Roveri, & Traverso, 1998; Jensen & Veloso, in press) and conditional planning (Peot & Smith, 1992; Borrajo & Veloso, 1996): instead of a plan representing a sequence of actions transforming a *single* initial state into a state fulfilling the top-level goals, DPlan constructs a planning tree or graph, representing optimal action sequences for *all* states belonging to the planning domain. A planning tree/graph represents the same information as a state-action table (Schoppers, 1987) but in a more compact way. Plan construction is based on breadth-first search and therefore works without backtracking. Because paths to nodes which are already covered (by shorter paths) in the plan are not expanded (similar

[1] We currently work on an extension of DPlan to function application ("updates"), thus that we can also deal with infinite domains. This work is done by Marina Müller in her diploma thesis.

to dynamic programming in A*), the algorithm is linear in the number of states.[2]

The general idea of DPlan is to construct a plan which represents a minimal spanning tree or DAG[3] (Christofides, 1975) of the (implicitly) given state space with the goal state(s) as root. In the current implementation, we present the complete set of states $\mathcal{D}$ as input. Planning problems are defined in the following way:

Definition 1 (Planning Problem) *A planning problem $\mathcal{P}(\mathcal{O}, \mathcal{D}, \mathcal{G})$ consists of a set of operators $\mathcal{O}$, a set of problem states $\mathcal{D}$, and a set of top-level goals $\mathcal{G}$. Operators are defined with preconditions φ and effects. Effects are given by ADD- and DEL-lists[4] (A, D). Conditioned effects are given by effect-preconditions φ_i and effects A_i, D_i. Currently, preconditions, effects and goals are restricted to conjunctions of positive literals and variables are existential quantified only. In contrast to other planners, domain $\mathcal{D}$ is not the set of constants occurring in a domain but the set of all possible states. A state is a conjunction of atoms (instantiated positive literals).*

An example for the specification of a planning problem from the *rocket* domain (Veloso & Carbonell, 1993) is given in table 2.1.[5] The planning goal is to transport two objects $O1$ and $O2$ from a place A to a destination B. The transport vehicle (*Rocket*) can only be moved in one direction (A to B). Therefore, it is important to load *all* objects before the rocket moves to its destination. The resulting universal plan is given in figure 2.1.

Plan *construction* is done by backward-operator application. A plan can be *executed* by forward application of all operators on the solution path from some initial state towards the root of the plan. Operator application is

[2]For optimal plan construction, it is not possible, to construct an algorithm with lower effort. For example, HPSr* (Haslum & Geffner, 2000) is exponential in the number of atoms (i. e. all instantiated literals in a domain!) to calculate a lower bound for the cost to reach a state containing a set of atoms.

[3]A DAG – directed acyclic graph – represents the "union" of all optimal plans for a problem, see chapter 3. Our notion of a planning graph – referring to minimal spanning trees or DAGs – is different from the planning graphs constructed by GRAPH-PLAN algorithms (Blum & Furst, 1997): There, a planning graph is a DAG over atoms. Furthermore, Graph-plan planning graphs do only represent a subset of possible states of a problem.

[4]More precise, ADD and DEL are *sets* of literals.

[5]Usually, a domain specification defines the operators (and maybe additional information as types and axioms) and a problem specification for a domain specifies top-level goals and an initial state. In the following, we will refer to operators and top-level goals as domain specification.

Table 2.1: The *Rocket* Domain

$\mathcal{D}$ = { ((at O1 B) (at O2 B) (at O3 B) (at Rocket B)),
((inside O1 Rocket) (at O2 B) (at O3 B) (at Rocket B)),
((inside O2 Rocket) (at O1 B) (at O3 B) (at Rocket B)),
((inside O3 Rocket) (at O1 B) (at O2 B) (at Rocket B)),
((inside O1 Rocket) (inside O2 Rocket) (at O3 B) (at Rocket B)),
((inside O1 Rocket) (inside O3 Rocket) (at O2 B) (at Rocket B)),
((inside O2 Rocket) (inside O3 Rocket) (at O1 B) (at Rocket B)),
((inside O1 Rocket) (inside O2 Rocket) (inside O3 Rocket) (at Rocket B)),
((at O1 A) (at O2 A) (at O3 A) (at Rocket A)),
((inside O1 Rocket) (at O2 A) (at O3 A) (at Rocket A)),
((inside O2 Rocket) (at O1 A) (at O3 A) (at Rocket A)),
((inside O3 Rocket) (at O1 A) (at O2 A) (at Rocket A)),
((inside O1 Rocket) (inside O2 Rocket) (at O3 A) (at Rocket A)),
((inside O1 Rocket) (inside O3 Rocket) (at O2 A) (at Rocket A)),
((inside O2 Rocket) (inside O3 Rocket) (at O1 A) (at Rocket A)),
((inside O1 Rocket) (inside O2 Rocket) (inside O3 Rocket) (at Rocket A)) }

$\mathcal{G}$ = {(at O1 B), (at O2 B), (at O3 B)} $\mathcal{O}$ = {load, move-rocket, unload} with

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

defined for fully-instantiated operators o (actions). In the case of operators with conditioned effects an instantiated operator is constructed for each possible effect. As usual, operator definitions are given in a set-theoretical way:

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

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

The DPlan algorithm will be presented in chapter 3. In the following, we will describe it informally: First, it is checked whether there is at least one state in the set of all states $\mathcal{D}$ which fulfills the top-level goals. If no state S with $\mathcal{G} \subseteq S$ exists, planning terminates without success. If one such state exists, this state is introduced as root of the universal plan. If more than one state in $\mathcal{D}$ fulfills $\mathcal{G}$, $\mathcal{G}$ is introduced as root node and all states fulfilling $\mathcal{G}$ are introduced as children with unlabeled edges (representing "empty" actions). All states introduced in the plan are removed from $\mathcal{D}$.

After constructing the root of the plan, DPlan proceeds recursively for each leaf calculating all immediate predecessor states. The predecessors are identified by backward operator application $Res_p^{-1}(S, \{o_1 \ldots o_n\})$ where $\{o_1 \ldots o_n\}$ is the set of all instantiated operators fulfilling the application condition for backward application (i. e., the Add-List of o_i matches a subset of S), and $Res_p^{-1}(S, \{o_1 \ldots o_n\})$ returns all predecessors of S. For each admissible predecessor S'_{ij}, with $o(S'_{ij}) = S_i$, o is introduced as edge and S'_{ij} as child node. The state set $\mathcal{D}$ is used for calculating predecessors in the following way: Free variables in an operator are instantiated only in accordance with the states in $\mathcal{D}$ – that is, the number of action candidates is restricted; only predecessors which are in $\mathcal{D}$ are accepted as admissible – that is, inconsistent states (which never were members of $\mathcal{D}$) as well as states which are already included on higher levels of the plan (already removed from $\mathcal{D}$) are omitted.

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

DPlan constructs optimal action sequences: For each state in the universal plan the path starting at this node and ending at the root gives the shortest possible sequence of actions transforming this state into the goal. If a state can be transformed by alternative action sequences into a goal state, the universal plan is a DAG (instead of a "real" tree). For the *rocket* problem with three objects, there are, for example, 3! different possible sequences to unload the objects at B which are all of the same (shortest possible) length (see fig. 2.1).

Using a predefined set of legal states ($\mathcal{D}$) reduces the planning problem to extracting the set of optimal plans. Alternatively, DPlan can con-

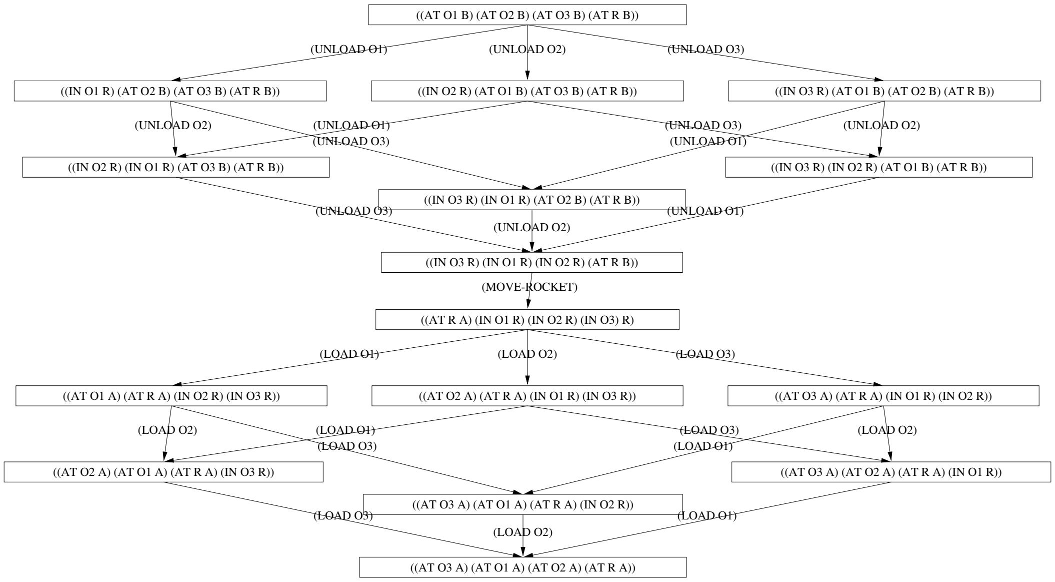


Figure 2.1: Optimal Universal Plan for *Rocket* (*in* = *inside*, *R* = *rocket*)

struct universal plans from a predefined goal *state* (i. e., a complete state description instead of a set of top-level goals) only. As a consequence, admissibility of planning steps cannot be checked via $\mathcal{D}$. Instead, we use $Res(Res^{-1}(S, o), o) \stackrel{?}{=} S$ as a criterium for admissibility (see chap. 3).

Alternatively to DPlan, each existing planning system (e. g., *Prodigy*, Veloso et al., 1995) can be used to construct minimal spanning DAGs by replacing the incorporated planning strategy (selection of one action for each planning step) by an "exploration mode" where all applicable actions are stored. Currently, some efficient universal planning systems are developed (MBP, Cimatti et al., 1998; UMOP, Jensen & Veloso, in press) which also could be used alternatively to DPlan. But remember, that we perform universal planning for *learning* recursive macros from some initial experience when solving a problem of small complexity. Human problem solvers also must invest much mental effort if they want to extract a general solution strategy – for example for solving the Tower of Hanoi problem (Klahr, 1978).

2.2 From Plans to Programs

2.2.1 Planning with Control Rules

Introducing control rules into planning has the advantage that the number of match-select-apply cycles gets reduced by giving guidance in which sequence planning goals are attacked. Providing the complete control structure of a domain, as in the form of a recursive program, eliminates search completely. For example, there might be a macro *load-all* which fulfills the goal {*(inside* ob_1 *Rocket), ... (inside* ob_n *Rocket)*} for all objects at some given place.

Application of a recursive program is encapsulated (Shavlik, 1990). That means, that until the program terminates no other primitive operators are checked for applicability. Thereby the number of match-select-apply cycles gets reduced by k if the *load* operator has to be applied k times. When a (set of) recursive programs represent the complete control knowledge of a domain (e. g., *load-all*, *move-rocket*, *unload-all*), the generated transformation sequences are optimal for each input state. If control knowledge covers only subproblems of a domain (e. g., only *load-all*), optimality cannot be guaranteed (c. f., Kalmar & Szepesvari, 1999): A recursive macro generates new goals for each application (e. g., loading of the next object) which cannot be interleaved with other pending goals (e. g., moving the rocket).

2.2.2 Learning Control Rules

If there exists knowledge about the structure of a planning domain, this knowledge can be incorporated into the domain specification by pre-defining control rules. The more interesting case is that such knowledge is acquired automatically from some initial planning experience. In the following, we will describe, how recursive macro-operators can be learned by exploring a problem of small complexity. For example, we can generate a plan for unloading three objects and generalize to *unload-all.* This is done by first transforming the initial plan into a finite program and than applying a generalization-to-n algorithm.

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

There are different approaches for learning recursive programs from examples. The most prominent of them are grammar inference (Sakakibara, 1997), genetic programming (Koza, 1992), inductive logic programming (ILP) (Muggleton & De Raedt, 1994; Flener & Yilmaz, 1999), and inductive synthesis of functional programs (Summers, 1977; Wysotzki, 1983; Le Blanc, 1994; Schmid & Wysotzki, 1998). We choose the "classical" functional approach for several reasons: Functional programs – in contrast to logic programs – represent control flow explicitly, that is, for learning control strategies, it is more straight-forward to infer *functional* programs. Genetic programming and most ILP approaches rely on search in hypothesis space (i. e., the set of syntactical correct programs or horn clauses) to generate a program which is complete (covers all positive examples) and consistent (covers no negative example) or a program satisfying some evaluation rule. Since a universal plan already contains crucial information about the structure of the searched for transformation rules, we prefer an approach which exploits this structure over (blind or heuristically guided) search. The classical approach offers a clear separation of (1) rewriting I/O examples to finite program terms and (2) generalization over these terms ("folding"). Approaches to grammar inference address this second step – inferring a recursive program from terms: a set of transformation rules (for generating a language) is inferred from a set of positive examples (words). While most grammar inference algorithms are proposed for regular grammars, recursive programs correspond to context-free tree grammars (Schmid et al., 1999). That is, the input examples (words) have to be terms.

While the second step of inductive program synthesis can be performed (nearly) by purely syntactical pattern matching, the first step is knowledge-dependent[6]. The result of rewriting I/O examples – i. e., the form and complexity of the finite program – is completely dependent on the background knowledge (here: predefined functions and predicates) provided for the rewrite-system. Additionally, the outcome depends on the used rewrite-

[6]Rewriting of I/O examples to terms corresponds roughly to the concept of "saturation" in inductive logic programming (Flener & Yilmaz, 1999).

strategy – i. e., even for a constant set of background knowledge rewriting can result in different programs. Theoretically, there are infinitely many possible ways to represent a finite program which describes how input examples can be transformed in the desired output. Because generalizability depends on the form of the finite program, this first step is the bottleneck of program synthesis. Here program synthesis is confronted with the crucial problem of AI and cognitive science – problem solving success is determined by the constructed representation (Kaplan & Simon, 1990).

We propose to use planning to realize rewriting of I/O examples. Inputs correspond to problem states, outputs to states fulfilling the desired goals, and transformation from input to output is realized by calculating the optimal action sequence. The resulting minimal spanning tree or DAG already represents the structure of the searched-for program, i. e., it gives an *ordering* of operator applications. Nevertheless, the plan cannot be generalized directly, but some transformation steps are needed to generate a finite program which than can be input to a generalization-to-n algorithm. We will address plan transformation in detail in chapter 4. For now, we will illustrate our approach with the *rocket* example.

2.2.3 Generating a Finite Program for the *Rocket*-Domain

Our general idea for plan transformation is to *infer the data type* which is underlying the universal plan. The idea to make planning more efficient by inferring "types" from plan structures which can be used as guideline when solving new planning tasks is also exploited by Long and Fox (2000). Before we can infer the underlying data structure for the *rocket* plan given in figure 2.1, the planning graph has to be **decomposed** into "uniform" sub-plans for *unload-all*, *move-rocket* (a single step), and *load-all* (see fig. 2.2).

Data type inference is done for each sub-plan. For both sub-plans (*unload-all* and *load-all*) there is a single root and a single leaf node and the sets of actions along all (six) possible paths from root to leaf are equal. From this observation we can conclude that the actual sequence in which the actions are performed is irrelevant, i. e., the underlying data structure of both sub-plans is a *set*. The (sub-) plan can be collapsed to one path. In chapter 4, we will propose that each plan can be captured by one of the basic data types *sequence*, *set*, or *list* or combinations (e .g. set of lists) of them; and we will give criteria from which these types can be inferred from the structure of the universal plan.

Associated with each data type are informations for how the initial value can be obtained and which constructor and selector functions have to be

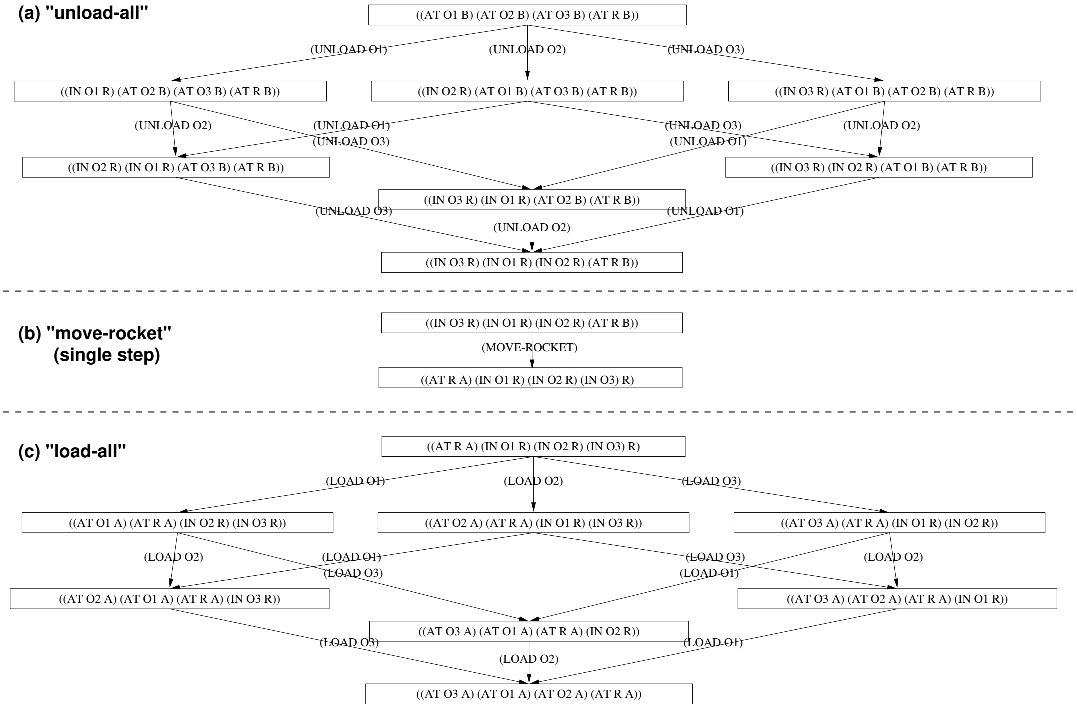


Figure 2.2: Sub-Plans of *Rocket*

introduced. For the *rocket* example the following steps are performed:

- The **initial set** is constructed by collecting all arguments of the actions along the path from root to leaf. That is, the initial value for the set can be $I = \{O1, O2, O3\}$ – when the left-most path of a sub-plan is kept. (If the involved operator has more than one argument, for example *(unload <obj> <place>)*, the constant arguments are ignored.)
- A "generalized" predicate – corresponding to the **empty-test** for a data type – is invented by generalizing over all literals in the root-node with an element of the initial set I as argument. That is, for the *unload-all* sub-plan, the new predicate is $p = (at^* \langle objSet \rangle\ B)$ with

$$(at^* \langle objSet \rangle\ B) = \begin{cases} true & if\ \forall o \in objSet:\ (at\ o\ B) \\ & holds\ in\ the\ current\ state \\ false & otherwise. \end{cases}$$

The original literals are replaced by p.

- The arguments of the generalized predicate are rewritten using the predefined **rest-selector**. We have the following replacements for the *unload* sub-plan (given from root to leaf):
  $(at^*\{O1, O2, O3\}\ B) == (at^*\{O1, O2, O3\}\ B)$
  $(at^*\{O1, O2\}\ B) \rightarrow (at^*(rst\{O1, O2, O3\})\ B)$
  $(at^*\{O1\}\ B) \rightarrow (at^*(rst(rst\{O1, O2, O3\}))\ B)$
  $(at^*\{\ \}\ B) \rightarrow (at^*(rst(rst(rst\{O1, O2, O3\})))\ B)$.
  The arguments of the actions are rewritten using the predefined **pick-selector**:
  $(unload\ O1) \rightarrow (unload(pick\{O1, O2, O3\}))$
  $(unload\ O2) \rightarrow (unload(pick(rst\{O1, O2, O3\})))$
  $(unload\ O3) \rightarrow (unload(pick(rst(rst\{O1, O2, O3\}))))$.
  Note, that we define *pick* and *rst* deterministical (e. g. as *head*/*tail* or *last* and *butlast*).

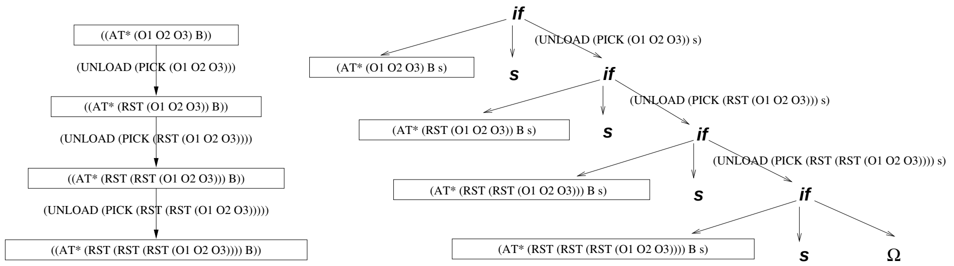
The transformed sub-plan for *unload-all* is given in figure 2.3.a.

Introduction of a data type is crucial for generating a generalizable program term. A program term represents the order of operator-application as well as the order of the *objects* of the program's domain (Manna & Waldinger, 1975). For example, a program over natural numbers as *factorial* incorporates the knowledge, that the numerical argument is reduced by one at each step and that 0 is the smallest element; a program over lists as *reverse* incorporates the knowledge, that the list is reduced by one element at each step and that the empty list is the smallest element (Summers, 1977). The universal plan gives us an order over operations, introducing a data type additionally provides an explicit representation of the order over the objects.

After introducing a data type into the plan, there is only one additional step necessary to interpret the plan as a program – **introducing a situation variable** (see c. f., Manna & Waldinger, 1987). Now the plan can be read as a nested conditional expression (see fig. 2.3.b).

While a planning algorithm applies an instantiated operator to a state currently in working-memory, interpretation of a (functional) programming term depends only on the instantiated parameters (values) of this term. Introducing a situation variable s in the plan makes it possible to treat a state as valuated parameter of an expression. That is, the primitive operators (*load*, *unload*, *move-rocket*) are now applied to the set of literals with which parameter s is currently instantiated.

Plan transformation for the *rocket* problem results in the following program structure



(a) (b)

Figure 2.3: Introduction of the Data Type *Set* (a) and Resulting Finite Program (b) for the *Unload-All* Sub-Plan of *Rocket* (Ω denotes "undefined")

```
(rocket oset s) = (unload-all oset (move-rocket (load-all oset s)))
```

where *unload-all* and *load-all* are recursive functions which were generated by our generalization-to-n algorithm from the corresponding finite programs. Note, that the parameters involve only the set of objects – this information is the only one necessary for *control*, while locations (A, B) and transport-vehicle (*Rocket*) are additional information necessary for the dynamics of plan construction.

2.3 Generalization-to-n

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

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

Definition 3 (Finite Program) *A finite program is a term* $t_S \in M(V, F \cup$

Ω). M is a term algebra over variables V and function symbols F with Ω as the undefined (empty) term.

Let us consider the finite program for unloading objects constructed above. Written as a term, this finite program is:

```
(IF (AT* OSET B S)
    S
  (UNLOAD (PICK OSET) S
          (IF (AT* (RST OSET) B S)
              S
            (UNLOAD (PICK (RST OSET)) S
                    (IF (AT* (RST (RST OSET)) B S)
                        S
                      (UNLOAD (PICK (RST (RST OSET))) S
                              (IF (AT*
                                   (RST (RST (RST OSET)))
                                   B
                                   S)
                                  S
                                OMEGA)))))))
```

with *M({oset,s},{b, at*, unload, pick, rst, if-then-else, Ω})*. The term can be valuated and interpreted in the usual way: *oset* can be instantiated by a set of objects – e. g., *(O1 O2 O3)* as in the planning problem above – s is a situation variable. Conditioned expressions are represented as *(if x y z)* with x as boolean expression, y as term which is evaluated if x is true and z as term which is evaluated if x is false. The constant b represents a location; the predicate *(at* oset b s)* was introduced in section 2.2.3 above. The operator *(unload o s)* corresponds to the operator defined in table 2.1, rewritten for situation calculus (Manna & Waldinger, 1987): instead of being applied to the current state in working memory, it is applied on the set of literals with which variable s is instantiated. The selector functions *pick* and *rst* are defined above. Symbol Ω (`OMEGA`) denotes the "undefined" term. For the *unload-all* program, the plan gives no information what to do if more than three objects are involved (*(AT* (RST (RST (RST OSET))) B S)* is not true).

Output of the generalization algorithm is a recursive program scheme:

Definition 4 (Recursive Program Scheme) *An RPS is a pair $\langle \Sigma, t\rangle$ with $\Sigma = \langle T_i(v_1 \ldots v_n) = t_i \mid i = 1 \ldots n\rangle$ is a system of equations ("subroutines") and $t \in M$ is the "main program"; t_i and t are elements of the extended term algebra $M(V, F \cup \Phi)$ with V as set of variables, F as set of function symbols* *(with $f^i \in F$ we denote functions with arity i), and Φ as set of function variables (names of user defined functions); $T_i \in \Phi$, and $v_1 \dots v_n \in V$.*

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

An RPS generalizing the *unload* term given above is $\Sigma = \langle$ *(unload-all oset s) = (if (at* oset B s) s (unload (pick oset) (unload-all (rst oset) s)))* $\rangle$ with $t =$ *(unload-all oset s)* for some constant set *oset* and some set of literals *s*.[7]

For program synthesis we reverse the idea of determining the semantic of a recursive function as its smallest fix-point (Wysotzki, 1983; Schmid & Wysotzki, 1998)[8]: from a given sequence of unfoldings we want to extrapolate the minimal recursive program scheme which can generate these unfoldings.[9]

Definition 5 (Folding of a finite program) *A finite program t_S can be folded into a recursive program scheme* iff *it can be decomposed into a sequence $\mathcal{T}^{(0)} = \Omega$, $\mathcal{T}^{(l)} = tr(\mathcal{T}^{(l-1)}_{[t/v]}/m)$ with $l = 1 \dots n$, $\mathcal{T}^{(n)} = t_S$ of partial transformations which successively cover a larger amount of inputs.*

For our example we have:

$$
\begin{array}{lcl}
\mathcal{T}^{(0)} & = & \Omega \\
\mathcal{T}^{(1)} & = & (if\ (at*\ oset\ b\ s)\ s\ (unload\ (pick\ oset)\ \Omega)) \\
\mathcal{T}^{(2)} & = & (if\ (at*\ oset\ b\ s)\ s\ (unload\ (pick\ oset)\ (if\ (at*\ (rst\ oset)\ b\ s)\ s\ (unload\ (pick\ (rst\ oset))\ \Omega)))) \\
\mathcal{T}^{(3)} & = & (if\ (at*\ oset\ b\ s)\ s\ (unload\ (pick\ oset)\ (if\ (at*\ (rst\ oset)\ b\ s)\ s\ (unload\ (pick\ (rst\ oset)) \\
 & & \quad (if\ (at*\ (rst(rst\ oset))\ b\ s)\ s\ (unload\ (pick\ (rst(rst\ oset)))\ \Omega)))))) \\
\mathcal{T}^{(4)} & = & (if\ (at*\ oset\ b\ s)\ s\ (unload\ (pick\ oset)\ (if\ (at*\ (rst\ oset)\ b\ s)\ s\ (unload\ (pick\ (rst\ oset)) \\
 & & \quad (if\ (at*\ (rst(rst\ oset))\ b\ s)\ s\ (unload\ (pick\ (rst(rst\ oset)))\ (if\ (at*\ (rst(rst(rst\ oset)))\ b\ s)\ s\ \Omega))))))) \\
 & = & t_S
\end{array}
$$

with $tr =$ *(if (at* oset b s) s (unload (pick oset) m))* and substitution $[(rst\ oset)\ /oset]$. Because $\mathcal{T}^{(l)} =$ *(if (at* oset b s) s (unload (pick oset)*

[7] For the abstract terms, as used in the definitions, we use the prefix-notation $f(x_i)$. For the concrete programs, we use a list-notation *(f x_i)*, representing control knowledge as LISP-programs.

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

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

$\mathcal{T}^{(l-1)}_{(rst\ oset)/oset]}$)) holds for all $\mathcal{T}$'s we can fold t_S and obtain the RPS given above[10].

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

Recursive functions for unloading and loading objects are:

```
(unload-all oset s) =
  (if (at* oset B s)
      s
      (unload (pick oset) (unload-all (rst oset) s))
  )
(load-all oset s) =
  (if (inside* oset Rocket s)
      s
      (load (pick oset) (load-all (rst oset) s))
  )
```

which are integrated in the "main" program `(rocket oset s)` given above. Further generalization over the constants – location (B) and vehicle (*Rocket*) – is possible. After parametrizing over these constants, the *load-all* and *unload-all* functions can be included in a class of transportation domains, as generalized control knowledge!

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

[10]For $\mathcal{T}^{(4)}$ the mapping is only partial because the last operator call is missing. But we allow that Ω maps every term.

Chapter 3

Universal Planning with DPlan

3.1 A Short History of DPlan

The original idea of using universal planning as initial step for inductive program synthesis was presented in Wysotzki (1987): sets of basic programs and axioms are used to construct a kind of conditional plan with top-level goals as root-node; ordering of dependent goals was realized by backtracking over plan construction. From 1995 to 1997 we explored (implemented and tested) several strategies for generating finite programs by planning: the approach proposed in Wysotzki (1987) was implemented in different versions by Ute Schmid, by Baback Paradian and by Olaf Brandes (see Parandian, Schmid, & Wysotzki, 1995; Schmid, Brandes, & Wysotzki, 1997).

Furthermore, we investigated combining forward search with decision tree learning: One approach is to generate optimal plans for each possible initial state of a domain with small complexity. Initial states then are represented as feature vectors where the set of all different literals occurring over states are used as features with value 1 if this literal occurs in a state description and value 0 otherwise. Each initial state is associated with the (or an) optimal action sequence for transforming it into a goal state. A decision tree algorithm (CAL2, see Unger & Wysotzki, 1981) is used to generate a classification program. Transforming such a program into a finite program fit for generalization-to-n involves the same problems as discussed above (see sect. 2.2) along with some additional problems: First, the decision tree does not necessarily represent an order over the number of transformations involved: it is possible that the first attribute already branches to a leaf representing the most complex transformation sequence. Second, there is no interleaving between actions and conditions which have to be fulfilled before executing these actions which is typical for recursion (see for example the finite program for *unload-all* in chap. 1), but instead, all conditions necessary to execute the complete transformation sequence are checked first (along a path in the decision tree) and the transformation sequence is executed completely afterwards. A possible way to split such compound actions is discussed in Wysotzki (1983)[1].

A second forward-search approach is presented in (Briesemeister et al., 1996).[2] Here an initial state is transformed into a goal state by forward search. Each state on the optimal path is associated with the following action and the state-action pairs are used to construct a decision tree with literals as features and (single) actions as leaves. A new problem is solved by feeding the initial state into the decision tree, executing the action given at the leaf of the path, thereby generating a new state, which again is input into the decision tree and so on, until a goal state is reached. If a state cannot be classified by the current decision tree, search is invoked again and the decision tree is expanded by incremental learning.

This approach is a good alternative to the one described in this report. The crucial difference between the universal planning/plan-transformation and the forward planning/decision-tree approach is that the first results in control knowledge represented as a set of recursive functions while the second results in control knowledge represented as state-features/action associations. Recursive functions represent the subgoal structure of a domain as well as the sequence of transformation steps in an explicit way, while feature/action associations represent the control flow indirectly, similar to production rules in a production system. The second approach has the advantage that learning can be performed incrementally – using example input states from problems of one domain –, while the first approach relies on the complete exploration of a domain for a problem of small size.

In 1998 we came up with the first version of DPlan as a state-based non-linear backward-planner constructing universal plans. This first algorithm, its formalization, proofs of completeness and correctness are presented in (Schmid, 1999). DPlan1.0 works for STRIPS-like domain specifications extended to binary conditioned effects. The generated universal plan is a minimal spanning tree. That is, for domains with sets of optimal solutions only

[1]This work is not documented in a paper. The documented program together with some examples can be obtained from Ute Schmid

[2]This approach is nearly identical to work of Martín and Geffner (2000).

one alternative is calculated for each possible state. DPlan1.0 was extended over the last year by several people in several ways:

- Domain specification in PDDL, STRIPS + general conditioned effects (see report of the student project "Extending DPlan To an ADL-subset" by Janin Toussaint, Ulrich Wagner, and Hakan Ilicik, 1999).
- Constructing universal plans without a predefined set of states (see report of the student project "Universal Planning without state sets" by Michael Christmann and Stefan Rönnecke, 1999).
- Preliminary work on plan construction using control knowledge represented as recursive macros (see report of the student project "Planning with recursive macros", Mischa Neumann and Ralf Ansorg, 1999).
- Extending DPlan to function application (Müller, 2000) – making it possible to plan problems involving manipulation of numbers (as water jug) and dealing with typical programming problems (as sorting of lists).

The modules of DPlan and the global data structure for storing plans are given in appendix A.

In this chapter we will present two aspects of DPlan which are not reported elsewhere: (a) a necessary restriction for state-based backward-planning, and (b) the extension of DPlan1.0 from generating minimal spanning trees to generating minimal spanning DAGs, representing the union of all optimal plans of a problem domain.

3.2 Completeness of Backward Planning

Universal planning necessarily has to rely on *backward* operator application, because no initial state is given. State-based backward planning has some inherent restrictions which we will describe in the following. We repeat the definitions for forward and backward operator application given in definition 2:

$$Res(S, o) = S \setminus D \cup A \ if \ \varphi \subseteq S$$

$$Res^{-1}(S, o) = S \setminus A \cup \{D \cup \varphi\} \ if \ A \subseteq S.$$

For forward-application, $\setminus D$ and $\cup A$ are only commutative for $A \cap D = \emptyset$[3]. When we delete literals before adding literals, as defined above, we guarantee that everything given in the ADD-list is really given for the new state.

Completeness and soundness of backward-planning means that the following diagram has to commute:

$$\begin{array}{lcl} S & == & Res(S', o) \\ \downarrow & & \uparrow \\ Res^{-1}(S, o) & == & S' \end{array}$$

If $Res^{-1}(S, o)$ results in $S' = Res(S', o)$ for all S, backward planning is sound. If for all S' with $Res(S', o) = S$ it holds that $Res^{-1}(S, o) = S$, backward planning is complete. In the following we will proof that these propositions hold with some restrictions.

For backward-planning to be sound, each sequence $S_{goal} \xrightarrow{o^{-1*}} S$ has to be a valid transformation sequence for S into a state fulfilling the top-level goals. That is, the following proposition has to hold for all S belonging to the domain: If $Res^{-1}(S, o) = S'$ then $Res(S', o) = S$. This only holds for some restrictions:

$$\begin{array}{rcl} Res(Res^{-1}(S, o), o) & \stackrel{?}{=} & S \\ Res(Res^{-1}(S, o), o) & = & \\ & = & Res^{-1}(S, o) \setminus D \cup A \ with\ \varphi \subseteq Res^{-1}(S, o) \\ & = & \{S \setminus A \cup \{D \cup \varphi\}\} \setminus D \cup A \\ & & \ with\ \varphi \subseteq \{S \setminus A \cup \{D \cup \varphi\}\} \ and\ A \subseteq S \\ & = & \{S \cup \{D \cup \varphi\}\} \setminus D \ with\ \varphi \subseteq \{S \setminus A \cup \{D \cup \varphi\}\} \\ & = & \{S \cup D\} \setminus D \ if\ \varphi \subseteq S \cup D \\ & = & S \ if\ S \cap D = \emptyset. \end{array}$$

The first restriction $\varphi \subseteq S \cup D$ means for forward-planning from S' to S, that φ holds after operator application if it is not explicitly deleted. In backward-planning φ is introduced as subgoals which have to hold in S'. This restriction seems unproblematic. The second restriction $S \cap D = \emptyset$ means for forward-planning from S' to S that S contains no literal from the DEL-list. For backward-planning all literals from the DEL-list are added

[3] In PDDL $A \cap D = \emptyset$ is always true: the effects are given in a single list with DEL-effects as negated literals. An expression `(and p (not p))` represents a contradiction!

and S has contained none of these literals. This restriction is in accordance with the definition of legal operators. Thus, soundness of DPlan is given. In general, soundness of backward-planning can be guaranteed by introducing an admissibility check for each constructed predecessor: For a new constructed state S' it can be checked that $S' = Res(S', o)$. If forward operator application does not result in S, the constructed predecessor is considered as not admissible and not introduced in the plan.

For backward-planning to be complete, we have to guarantee that we find a valid transformation sequence if such a sequence exists. For our universal planner that means, that plan Θ must contain all states S which can be transformed to a state fulfilling the top-level goals. That is, the following proposition has to hold: If $Res(S', o) = S$ then $Res^{-1}(S, o) = S'$. This only holds for some restrictions:

$$\begin{aligned}
Res^{-1}(Res(S', o), o) &\stackrel{?}{=} S' \\
Res^{-1}(Res(S', o), o) &= \\
&= Res(S', o) \setminus A \cup \{D \cup \varphi\} \ with \ A \subseteq Res(S', o) \\
&= (S' \setminus D \cup A) \setminus A \cup \{D \cup \varphi\} \\
&\quad\ with \ A \subseteq (S' \setminus D \cup A) \ and \ with \ \varphi \subseteq S' \\
&= S' \setminus D \cup \{D \cup \varphi\} \ with \ \varphi \subseteq S' if S' \cap A = \emptyset \\
&= S' \cup \varphi \ with \ \varphi \subseteq S' \ if \ D \subseteq S' \\
&= S'.
\end{aligned}$$

The first restriction $S' \cap A = \emptyset$ means that DPlan can only consider states which does not already contain something which is added by an applicable operator. The second restriction $D \subseteq S'$ means that DPlan can only consider states which contain all literals which are deleted by an applicable operator. While this is a real source of incompleteness, we are still looking for a meaningful domain where these cases occur. As long as DPlan works on $\mathcal{D}$ as set of all states of a domain, incompleteness can be overcome – with a loss of efficiency – by constructing predecessors in the following way:

$$Res^{-1}(S, o) = S \setminus A^* \cup \{D^* \cup \varphi\}$$

for all combinations of subsets A^* of A and D^* of D if $A \subseteq S$.

Thus, all possible states containing subsets of D and A with the special case of inserting all literals from D and deleting all literals from A, would be constructed and admissibility could be checked as usual via occurrence in $\mathcal{D}$. If working with $\mathcal{D}$ as set of constants of the domain (i. e., without a predefined state set), admissibility can only be checked by introducing domain axioms.

With the introduction of arbitrary conditional effects as allowed in PDDL, additional restrictions arise.[4]

Currently, we feel no need to extend our algorithm to overcome the restrictions given above, because these restrictions hold for a lot of benchmark problems, as for example used in the last planning competition (AIPS 2000). A planner which also uses backward search with similar restrictions is HSPr* (Haslum & Geffner, 2000): For $S \cap A \neq \emptyset$ and $S \cap D = \emptyset$: $Res^{-1}(S, o) = S \setminus A \cup \varphi$.

3.3 Universal Plans as Sets of Optimal Plans

In the following we will introduce the basic concepts for constructing optimal universal plans.[5]

3.3.1 Union of Optimal Plans

Definition 6 (Plan) *An plan for transforming a state S_0 into a state S_n fulfilling the top-level goals is defined as a transformation sequence $(S_0, S_1) \ldots (S_i, S_{i+1}) \ldots (S_{n-1}, S_n)$ with $\Pi = \{S_0, \ldots S_n\}$ as set of all states contained in the plan and $S_i \leq S_j$ as order over Π where $S_i < S_j$ holds if S_i occurs before S_j in the plan. The distance between two states in the plan $l(S_i, S_j)$ is defined as length of the transformation sequence from S_i to S_j. With $p(S_i) \in N$ we denote the "level" at which a state occures in the plan. The relation $S_i < S_j$ describes a total order over Π: there is a smallest element S_0, the relation is antisymmetric and transitive and for each S_i with $i = 0 \ldots n-1$ there exists an S_j with $S_i < S_j$. In the following, Π denotes a plan, that is, a set of states together with the order relation induced by the transformation sequence.*

Definition 7 (Optimal Plan) *A plan Π is optimal, if for each pair of transformations $(S_i, S_m), (S_m, S_j)$ there does not exist a state $S_k \neq S_m$ with $(S_i, S_k), (S_k, S_j)$ where $l(S_i, S_k) < l(S_i, S_m)$ or $l(S_k, S_j) < l(S_m, S_j)$. In the following, Π refers to optimal plans if not stated otherwise and $<_\Pi$ denotes the ordering over Π.*

---

[4] These restrictions are discussed in the report of the student project by Janin Toussaint, Ulrich Wagner, and Hakan Ilicik.

[5] The concepts and notations are based on (Mädler, 1992).

For optimal univeral planning we want to construct the union of all optimal plans for transforming arbitrary states into a state fulfilling the top-level goal. In the following we interpret a sequence $(S_0, S_1) \rightarrow \ldots (S_i, S_{i+1}) \rightarrow \ldots (S_{n-1}, S_n)$ as a backward-chain where S_0 is a state fulfilling the top-level goals. In general, there might be different states fulfilling the top-level goals. The following definitions are restricted to unique goal-states for reasons of clarity. In the DPlan algorithm we work on an extension of these definitions discussed below. An explicit enumeration of all optimal plans for the *rocket* domain with two objects is given in table 3.1.

Definition 8 (Union of Optimal Plans) *A set Π^* is the union of all optimal plans Π_i starting at a fixed node S_0 if the following propositions hold:*

- *The elements of each optimal plan Π_i from S_0 to some arbitrary state S_n are elements of Π^*.*
- *If $S_i < S_j$ holds for Π_i then it also holds for Π^*.*
- *$l_{\Pi_i}(S_i, S_j) = l_{\Pi^*}(S_i, S_j)$ for all $S_i, S_j \in \Pi_i$.*

The union of the total ordered sets Π_i with S_0 as smallest element results in a complete partial order $<_{\Pi^*}$ over Π^*: $S_0 < S_i$ forall $S_i \in \Pi^*$ (existance of a least element), and for each increasing sequence $S_1 < S_2 < \ldots < S_i$ in Π^* there exists a least upper bound (completeness). (The order is partial because there can be elements in Π^* which only occur in a single optimal plan Π_i, that is, no relation between these elements and elements of other optimal plans is defined.)

A consequence from these propositions is, that if some S_i occurs at level $p(S_i)$ in an optimal plan Π_i then it occurs at exactly the same level in Π^*. That is, it is not possible that another optimal plan Π_j from S_0 to S_i exists where S_i is reached earlier or later. In the first case, Π_i would not be an optimal plan, in the second case Π_j would not be an optimal plan. The union of optimal plans is illustrated in figure 3.1. The union of all optimal plans for the *rocket* domain with three objects is given in figure 2.1 in chapter 1.

3.3.2 Unions of Optimal Plans are DAGs

Another perspective on optimal universal plans can be gained by using graph theoretical concepts: A problem domain implies a problem graph (structure of the state space). We can show that the union of all optimal plans for a given goal-state corresponds to a "minimal spanning" directed acyclic graph

Table 3.1: The Set of All Optimal Plans for the *Rocket* Domain

Π_1 = ((at O1 B) (at O2 B) (at Rocket B))

Π_2 = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (at O2 B) (at Rocket B))

Π_3 = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O2 Rocket) (at O1 B) (at Rocket B))

Π_4 = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B))

Π_5 = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O2 Rocket) (at O1 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B))

Π_6 = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B)) $\xleftarrow{(move-rocket)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket A))

Π_7 = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O2 Rocket) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B)) $\xleftarrow{(move-rocket)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket A))

Π_8 = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B)) $\xleftarrow{(move-rocket)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket A)) $\xleftarrow{(loadO1A)}$ ((at O1 A) (inside O2 Rocket) (at Rocket A))

Π_9 = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O2 Rocket) (at O1 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B)) $\xleftarrow{(move-rocket)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket A)) $\xleftarrow{(loadO1A)}$ ((at O1 A) (inside O2 Rocket) (at Rocket A))

Π_{10} = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B)) $\xleftarrow{(move-rocket)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket A)) $\xleftarrow{(loadO2A)}$ ((at O2 A) (inside O1 Rocket) (at Rocket A))

Π_{11} = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O2 Rocket) (at O1 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B)) $\xleftarrow{(move-rocket)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket A)) $\xleftarrow{(loadO2A)}$ ((at O2 A) (inside O1 Rocket) (at Rocket A))

Π_{12} = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B)) $\xleftarrow{(move-rocket)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket A)) $\xleftarrow{(loadO1A)}$ ((at O1 A) (inside O2 Rocket) (at Rocket A)) $\xleftarrow{(loadO2A)}$ ((at O1 A) (at O2 A) (at Rocket A))

Π_{13} = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O2 Rocket) (at O1 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B)) $\xleftarrow{(move-rocket)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket A)) $\xleftarrow{(loadO1A)}$ ((at O1 A) (inside O2 Rocket) (at Rocket A)) $\xleftarrow{(loadO2A)}$ ((at O1 A) (at O2 A) (at Rocket A))

Π_{14} = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B)) $\xleftarrow{(move-rocket)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket A)) $\xleftarrow{(loadO2A)}$ ((at O2 A) (inside O1 Rocket) (at Rocket A)) $\xleftarrow{(loadO1A)}$ ((at O1 A) (at O2 A) (at Rocket A))

Π_{15} = ((at O1 B) (at O2 B) (at Rocket B)) $\xleftarrow{(unloadO2B)}$ ((inside O2 Rocket) (at O1 B) (at Rocket B)) $\xleftarrow{(unloadO1B)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket B)) $\xleftarrow{(move-rocket)}$ ((inside O1 Rocket) (inside O2 Rocket) (at Rocket A)) $\xleftarrow{(loadO2A)}$ ((at O2 A) (inside O1 Rocket) (at Rocket A)) $\xleftarrow{(loadO1A)}$ ((at O1 A) (at O2 A) (at Rocket A))

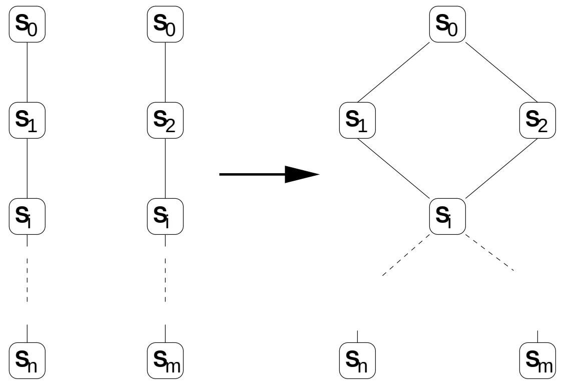


Figure 3.1: Union of Optimal Plans

(MSDAG), which can be extracted from the problem graph. We introduce the concept of an MSDAG as an extension of the concept of minimal spanning trees (Dijkstra, 1959).

Definition 9 (Problem Graph) *A problem graph $G = (V, R)$ is given by a set of nodes V representing problem states, edges $R \subseteq V \times V$ representing operator applications. For an edge $(v_i, v_j) \in R$ mappings α, ω: $R \to V$ give the starting/ending node: $\alpha(v_i, v_j) = v_i$, $\omega(v_i, v_j) = v_j$.*

When only presented with a set of top-level goals, a set of initial states, and a set of operators (as usual in planning and problem solving), the problem graph is only implicitely given. But each application of an operator results in a node-edge-node path which is contained in this problem graph. The same is true for universal planning (where no initial states are given). As an example, the problem graph for the *tower* problem[6] is given in figure 3.2.

Definition 10 (Minimal Spanning Tree) *A minimal spanning tree T_* of a problem graph $G = (V, R, \alpha, \omega)$ is defined as*

- *$T_* = (W, S)$ is a partial graph of G, $T_* \subseteq G$ with $W = V$ and $S \subseteq R$.*
- *T_* is a tree (that is, a directed acyclic graph where each node has maximally one predecessor).*
- *$\beta : W \to \mathcal{R}$ is an evaluation of edges $r \in R$. The β-value of T_* is defined as $\beta(T_*) = \sum_{r \in T_*} \beta(r)$. T_* is a minimal spanning tree if*

[6] The *tower* problem will be discussed in detail in chapter 4.

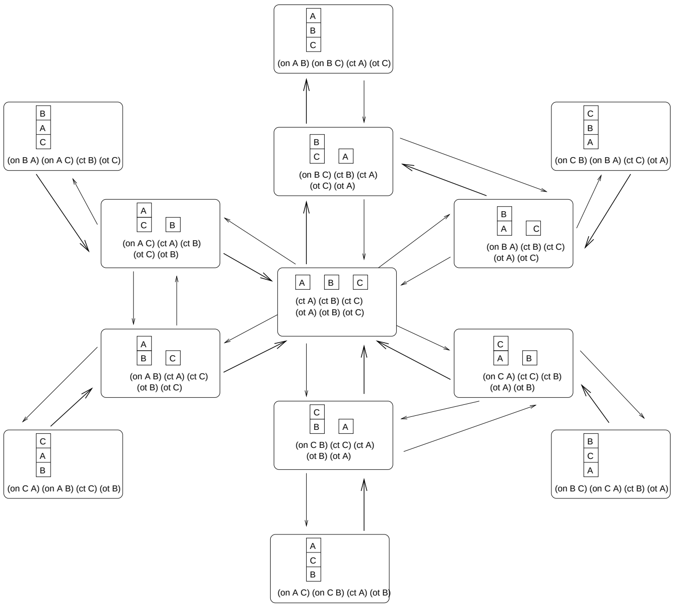


Figure 3.2: Problem Graph for the *Tower* Problem

> $\beta(T_*) = min\{\beta(T) \mid T \textit{ is spanning tree of } G\}$.
> *For the special case of unevaluated edges (each edge has identical costs), we have:* $\beta(T_*) = \|S\|$ *and a spanning tree is always a minimal spanning tree.*

In the following we add the following restriction to this definition:

Definition 11 (Minimal Spanning Tree with Fixed Root) *There exists a fixed node* S_0 *in* G *corresponding to the goal-state and a minimal spanning tree of* G *has to represent this state as root.*

We can write (minimal spanning) trees as lists in the following way:

Definition 12 (Representing Trees as Lists) *A tree can be represented as*

$$tree = leaf \mid node(edge_1(tree) \ldots edge_n(tree)).$$

In our case, $leaf$ *and* $node$ *represent problem states and* $edge$ *actions. We define* $x \in tree$ *to be true if* x *is contained in the set of nodes of a tree and false otherwise.*

The given restriction on minimal spanning trees reduces the number of minimal spanning trees of a problem graph. A minimal spanning tree with fixed root can be extracted from an (implicitly given) problem graph by the following procedure:

Algorithm 1 (Construction of a Minimal Spanning Tree Θ)

- *Use the goal state as root of* Θ
- *Recurse for each leaf* S *of* Θ
  - *Calculate the set of all different predecessor states* $\{S_i \mid Res^{-1}(S, o_i) = S'_i\}$ *of* S *with* $S'_i \notin \Theta$
  - *Extend* S *to* $S(o_1(S'_1) \ldots o_n(S'_n))$
- *Terminate (for each leaf* S*) if* $\{S_i \mid Res^{-1}(S, o_i) = S'_i\} \subseteq \Theta$ *or if* $\{S_i \mid Res^{-1}(S, o_i) = S'_i\} = \emptyset$.

Algorithm 1 constructs optimal plans because for each node only such predecessors are included in Θ which are not already contained in Θ and because the algorithm is based on a breadth-first strategy.

If there is no unique minimal spanning tree with fixed root for a problem, algorithm 1 constructs only a subset of all optimal plans. As an example, see figure 3.3 showing one of 36 possible minimal spanning trees for the *rocket* problem with three objects (3! possible sequences of unloading combined with 3! possible sequences of loading).

Lemma 1 (Minimal Spanning Tree as Subset of Π^*)
For a given problem, the set of nodes of the minimal spanning tree Θ *is identical to the set of nodes in* Π^*. *The edges* (S_i, S_j) *(represented as* $S(o(S'))$ *in alg. 1) define an order* $S_i < S_j$ *and* $\forall (S_i, S_j) \in \Theta : (S_j, S_j) \in \Pi^*$. *But there might exist relations* $S_i < S_j$ *in* Π^* *which are not given in* Θ. *That is* $\Theta = \Pi^*$ *and* $\{(S_i, S_j) \in <_\Theta\} \subseteq \{(S_i, S_j) \in <_{Pi^*}\}$.

Proof 1 (Lemma 1)

- $|\Pi^*| = |\Theta|$ *(*Θ *contains exactly the nodes in* Π^**):*
  Π^* *contains all states of a problem domain which are elements of optimal plans starting from a goal state.* Θ *is constructed inductively by alg. 1. It contains the state fulfilling the top-level goals (instantiation) and it all states which are reachable from these states (recursion) in accordance with the definition of* $Res^{-1}(S, o)$.

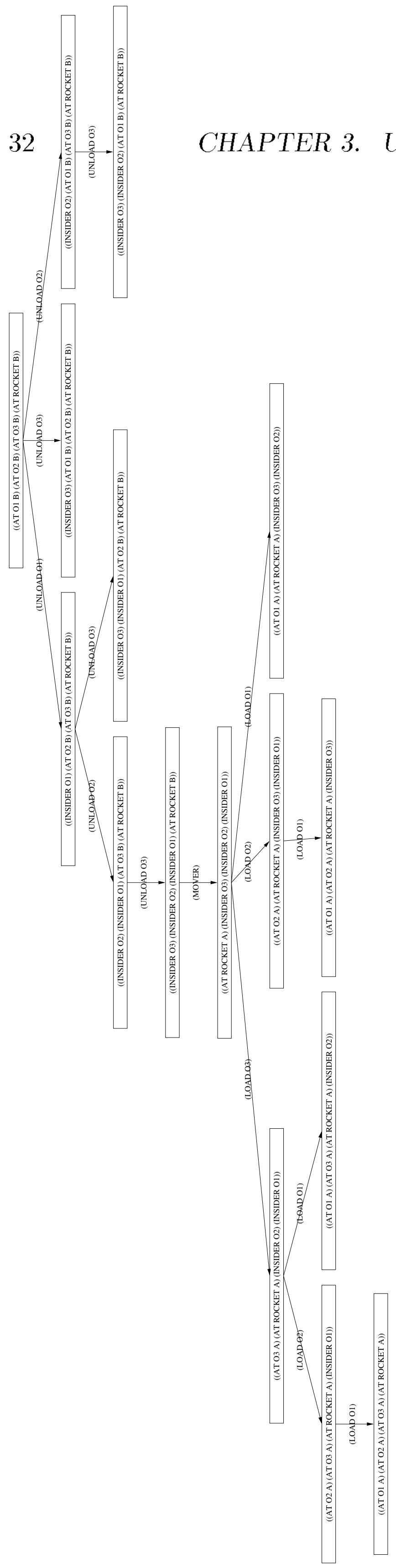


Figure 3.3: A Minimal Spanning Tree for the *Rocket* Domain

- $(S_i, S_j) \in <_\Theta \rightarrow (S_i, S_j) \in <_{\Pi^*}$ *(The order of states implied by Θ corresponds to the order of states with respect to their level in optimal plans):*
  follows from the construction of Θ by alg. 1.

- *For a leaf node S in the minimal spanning tree Θ there might exist a $Res^{-1}(S, o) = S'$ but no edge from S to S'. If S' is already contained in an optimal plan with $l(S') \leq l(S)$ then $(S, S') \notin <_{\Pi^*}$. But, if $l(S') = l(S) + 1$ then (S, S') is contained in $<_{\Pi^*}$ but not in $<_\Theta$ (no edge from S to S')!*

The extension of algorithm 1 to construct the union of *all* optimal plans of a domain as defined in definition 8 is obvious: If $Res^{-1}(S, o) = S'$ with $l(S') = i$ and S' is already in Θ as predecessor to some other S_p at level i, than an edge from S to S' is introduced. Again, the argumentation for sets of optimal plans holds: If S' would already be contained at a higher level $j < i$ in Θ, than $Res^{-1}(S, o) = S'$ does not belong to an optimal plan! Because now there are edges (S_p, S') and (S, S') in Θ, Θ is no longer a tree, but a graph! This graph is still directed (from the goal state as root to predecessors) and contains no cycles.

Definition 13 (Minimal Spanning DAG) *A MSDAG D_* of a problem graph $G = (V, R, \alpha, \omega)$ is defined as*

- *$D_* = (W, S)$ is a partial graph of G, $D_* \subseteq G$ with $W \subseteq V$ and $S \subseteq R$.*

- *D_* is a directed acyclic graph (that is, it is weakly connected and it contains no path with a cycle).*
  Furthermore, for D_* the following restriction holds: For each edge $(w, w') \in S, l(w) + 1 = l(w')$.

- *$\beta : W \rightarrow \mathcal{R}$ is an evaluation of edges $r \in R$. The β-value of D_* is defined as $\beta(D_*) = \sum_{r \in D_*} \beta(r)$. D_* is a MSDAG if $\beta(D_*) = min\{\beta(D) \mid \text{ D is directed and acyclic partial graph of G } \}$.*
  For the special case of unevaluated edges (each edge has identical costs), we have: $\beta(D_*) = \|S\|$.

- *Again, we restrict MSDAGs to a fixed root-node, corresponding to the goal-state of a problem domain.*

We can write (minimal spanning) DAGs as lists in the following way:

Definition 14 (Representing DAGs as Lists) *A DAG can be represented as*

$$dag = leaf \mid node(edge_1(dag_1) \dots edge_n(dag_n)).$$

In our case, leaf and node represent problem states and edge actions. For the list representation, a node can be contained more than once at a fixed level of the DAG, that is, the roots of some dag_i may be identical. We define $x \in dag$ to be true if x is contained in the set of nodes of a DAG and false otherwise.

Now algorithm 1 can be modified to constructing an MSDAG:

Algorithm 2 (Construction of a MSDAG Θ)

- *Use the goal state as root of Θ*
- *Recurse for each leaf S of Θ*
  - *Calculate the set of all different predecessor states $\{S_i \mid Res^{-1}(S, o_i) = S_i'\}$ of S with $S_i' \notin \Theta$ or $l(S_i') = l(Res^{-1}(S, o_i))$*
  - *Extend S to $S(o_1(S_1') \dots o_n(S_n'))$*
- *Terminate (for each leaf S) if $\{S_i \mid Res^{-1}(S, o_i) = S_i'\} \subseteq \Theta$ or if $\{S_i \mid Res^{-1}(S, o_i) = S_i'\} = \emptyset$.*

In contrast to a minimal spanning tree, the MSDAG constructed by algorithm 2 is unique and corresponds to the set of optimal plans Π^*.

Lemma 2 (Correspondence of MSDAG and Π^*)
For a given problem, the set of nodes of the MSDAG Θ is identical to the set of nodes in Π^*. The edges (S_i, S_j) (represented as $S(o(S'))$ in alg. 2) define an order $S_i < S_j$ and $\forall (S_i, S_j) : (S_i, S_j) \in \Theta \Leftrightarrow (S_j, S_j) \in \Pi^*$.

Proof 2 (Lemma 2)

- $|\Pi^*| = |\Theta|$ *(Θ contains exactly the nodes in Π^*):*
  Π^* contains all states of a problem domain which are elements of optimal plans starting from a goal state. Θ is constructed inductively by alg. 2. It contains the state fulfilling the top-level goals (instantiation) and it all states which are reachable from these states (recursion) in accordance with the definition of $Res^{-1}(S, o)$.

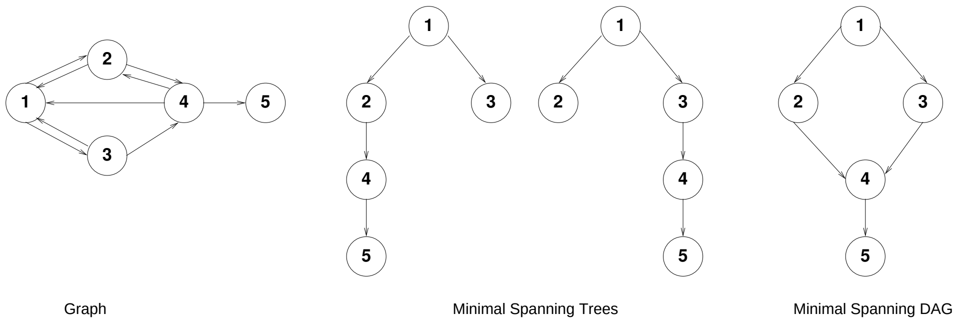


Figure 3.4: A Graph, Its Minimal Spanning Trees, and Its Minimal Spanning DAG with Node 1 as Predefined Root

- $(S_i, S_j) \in <_\Theta \rightarrow (S_i, S_j) \in <_{\Pi^*}$ *(The order of states implied by* Θ *corresponds to the order of states with respect to their level in optimal plans):*
  follows from the construction of Θ *by alg. 2.*

- $(S_i, S_j) \in <_\Theta \Leftrightarrow (S_i, S_j) \in <_{\Pi^*}$*:*
  If $(S_i, S_j) \in <_\Theta$ *then* $Res(S_i, o) = S_j$ *belongs to an optimal action sequence because of the construction of* Θ *by algorithm 2 and* $l(S_i) + 1 = l(S_j)$ *holds in* Π^**.*
  If $l(S_i) + 1 = l(S_j)$ *holds in* Π^* *then* $(S_i, S_j) \in <_\Theta$ *because of the construction of* Θ *by algorithm 2.*

If there exists a unique minimal spanning tree with the goal state(s) as root for the problem graph underlying a planning domain, the minimal DAG corresponds to this tree; if there exists a set of minimal spanning trees with the goal state(s) as root, the DAG represents the set-theoretical union of these trees with the restriction that identical states have to occur at identical levels of the minimal spanning trees which are unified. An example is given in figure 3.4. Because we construct plans starting with the goal state(s), the root of a MSDAG is fixed.

3.3.3 The DPlan Algorithms

In the following, a more detailed description of the algorithms for constructing sets of optimal plans is given. An extended representation of algorithm 1 for constructing a minimal spanning tree is given as algorithm 3, an extended representation of algorithm 2 for constructing a minimal spanning

DAG is given as algorithm 4. For algorithm 3, a full formalization, proofs of termination, soundness, and optimality are given in Schmid (1999).[7]

Algorithm 3 (DPlan MST-Algorithm 1)

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

1. *Initialization (root of* Θ*):*
   Let $S_\mathcal{G} = \{S_i \mid i = 0 \ldots n; S_i \in \mathcal{D} \text{ and } \mathcal{G} \subseteq S_i\}$
   - *if* $S_\mathcal{G} = \emptyset$: $\Theta = \perp$
   - *if* $\|\mathcal{S}_\mathcal{G}\| = 1$: $\Theta = S$ *for* $S_\mathcal{G} = \{S\}$, $\mathcal{D} = \mathcal{D} \setminus S_\mathcal{G}$
   - *else (more than one state fulfilling the top-level goals):* $\Theta = \mathcal{G}(S_1 \ldots S_n)$ *(a term/tree with the top-level goals as root and all states fulfilling* $\mathcal{G}$ *as children) for* $S_\mathcal{G} = \{S_1, \ldots, S_n\}$, $\mathcal{D} = \mathcal{D} \setminus \mathcal{S}_\mathcal{G}$
2. *Tree expansion (if* $\Theta \neq \perp$*):*
   For all leafs S_i *of* Θ*: calculate all* different *predecessors* S'_{ij} *with* $Res^{-1}(S_i, o_{ij}) = S'_{ij}$ *and* $S'_{ij} \in \mathcal{D}$[8] *where* o_{ij} *is an instantiated operator from* $\mathcal{O}$
   $\Theta(S_i) = S_i(o_{i1}(S'_{i1}) \ldots o_{in}(S'_{in}))$ *(a sub-term/sub-tree where node* S_i *has children* $S'_{i1} \ldots S'_{in}$ *with the connecting edges labeled with actions* $o_{i1} \ldots o_{in}$*),* $\mathcal{D} = \mathcal{D} \setminus \{S'_{ij}\}$
3. *Termination:* $\mathcal{D} = \emptyset$ *or no leaf in* Θ *can be expanded*

Because the algorithms only differ at one (crucial!) step, we can describe them together: Input in the algorithm is a planning problem as defined in definition 1, output is a minimal spanning tree or the MSDAG of a problem. Remember, that the minimal spanning tree of a problem might not be unique, that is, the output is only a subset of the set of optimal plans Π^* while the MSDAG corresponds to the set of optimal plans. First, it is checked whether there is at least one state in the set of all states $\mathcal{D}$ which fulfills the top-level goals. If no state S with $\mathcal{G} \subseteq S$ exists, planning terminates without success. If one such state exists, this state is introduced as root of the plan. If more than one state in $\mathcal{D}$ fulfills $\mathcal{G}$, $\mathcal{G}$ is introduced as root node and all states fulfilling $\mathcal{G}$ are introduced as children with unlabeled edges (representing "empty" actions). All states introduced in the plan (that is for the first step all states fulfilling the goal) are removed from $\mathcal{D}$.

Algorithm 4 (DPlan MSDAG-Algorithm)

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

[7] The termination proof given there holds only for a restricted set of domains, see sect. 3.2.

[8] Calculation of different predecessors is guaranteed by removing all children of a state S_i immediately from $\mathcal{D}$, that is, before the children of the next state on the same tree-level are calculated.

- *Output: minimal spanning DAG Θ for $\mathcal{P}$, or $\bot$*

1. *Initialization (root of Θ):*
   Let $S_{\mathcal{G}} = \{S_i \mid i = 0 \ldots n; S_i \in \mathcal{D}$ and $\mathcal{G} \subseteq S_i\}$
   - *if $S_{\mathcal{G}} = \emptyset$: $\Theta = \bot$*
   - *if $\|\mathcal{S}_{\mathcal{G}}\| = 1$: $\Theta = S$ for $S_{\mathcal{G}} = \{S\}$*
   - *else (more than one state fulfilling the top-level goals): $\Theta = \mathcal{G}(S_1 \ldots S_n)$ (a term/tree with the top-level goals as root and all states fulfilling $\mathcal{G}$ as children) for $S_{\mathcal{G}} = \{S_1, \ldots, S_n\}$*
2. *DAG expansion (if $\Theta \neq \bot$):*
   For all different leafs S_i of Θ: calculate all predecessors S'_{ij} with $Res^{-1}(S_i, o_{ij}) = S'_{ij}$ and $S'_{ij} \in \mathcal{D}$ where o_{ij} is an instantiated operator from $\mathcal{O}$
   $\Theta(S_i) = S_i(o_{i1}(S'_{i1}) \ldots o_{in}(S'_{in}))$ (a sub-term/sub-tree where node S_i has children $S'_{i1} \ldots S'_{in}$ with the connecting edges labeled with actions $o_{i1} \ldots o_{in}$)[9] $\mathcal{D} = \mathcal{D} \setminus \{S'_{ij}\}$
3. *Termination: $\mathcal{D} = \emptyset$ or no leaf in Θ can be expanded*

After constructing the root node, DPlan proceeds recursively for each leaf calculating all immediate predecessor states. The predecessors are identified by backward operator application $Res_p^{-1}(S, \{o_1 \ldots o_n\})$ where $\{o_1 \ldots o_n\}$ is the set of all instantiated operators fulfilling the application condition for backward application (that is, the Add-List of o_i matches a subset of S), and $Res_p^{-1}(S, \{o_1 \ldots o_n\})$ returns all potential predecessors of S. In general, a predecessor S'_{ij} is admissible if $o(S'_{ij}) = S_i$. In DPlan, admissibility is checked via look-up in the state-set $\mathcal{D}$. After a state is accepted as admissible predecessor and included in the plan, it is removed from $\mathcal{D}$.

The algorithms for constructing a universal plan (alg. 3) vs. a MSDAG (alg. 4) differ on the time when a predecessor is removed from $\mathcal{D}$. For algorithm 3, a state S'_{ij} is removed from $\mathcal{D}$ *immediately* after it is included in the plan. For algorithm 4 all predecessors for all leafs of the current plan are calculated and removed *afterwards*. Therefore, identical states might occur at the same level of the plan which are "unified" to one node. That is, when expanding the next level of the plan only different states are regarded.

The algorithm terminates successfully with a minimal spanning tree or DAG if $\mathcal{D}$ is empty. If $\mathcal{D}$ is not empty and nevertheless there is no operator applicable which leads to a state in the set of remaining states $\mathcal{D}$, the graph (problem space) underlying the domain $\mathcal{D}$ might be disconnected or contain uni-directional edges.

[9] There can be identical S'_{ij} on a given level of Θ which were produced from the same parent state with a different action or from different parent states. A state which occurs more than once is expanded only once. $\mathcal{D}$ is reduced after a level of Θ is expanded completely.

Chapter 4

Transforming Plans into Programs

4.1 Transformation and Type Inference

Now we come back to our central goal – learning domain specific control knowledge from plans. Control knowledge is represented as sets of recursive functions (recursive program schemes). Our general approach – as outlined in chapter 2 – is, to explore a domain by universal planning, to transform the plan into a (set of) finite program(s) and to generalize over the finite program(s) (see fig. 4.1). While plan construction and generalization-to-n are straight-forward and can be dealt with by domain-independent, generic algorithms, plan transformation is knowledge dependent and therefore the bottleneck of our approach.

Starting point for plan transformation is a *complete* and *correct* universal plan. The result of plan transformation is a finite program term. The completeness and correctness of this program can be checked by using each state in the original plan as input to the finite program and check (1) whether the interpretation of the program results in the goal state and (2) whether the number of operator applications corresponds to the number of edges on the path from the given input state to the root of the plan. Of course, because generalization-to-n is an inductive step, we cannot guarantee the completeness and correctness of the inferred recursive program. The recursive program could be empirically validated by presenting arbitrary input states of the domain.

In chapter 2 we gave the motivation for plan transformation. In the following we will describe the method in detail. Plan transformation consists

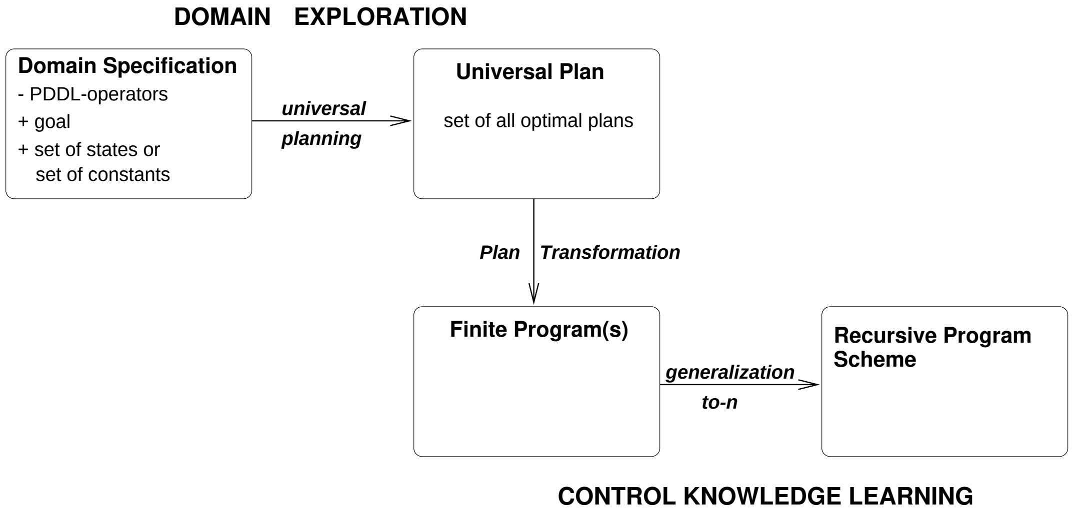


Figure 4.1: Induction of Recursive Functions from Plans

of three steps, which we will describe below:

Plan Decomposition: If a universal plan consists of parts with non-overlapping actions, the plan is splitted into sub-plans.
In this case, the following transformation steps are performed for each sub-plan separately and a term giving the structure of function-calls is generated from the decomposition structure.

Data Type Inference: The ordering underlying the objects involved in action execution is generated from the structure of the plan. From this order, the data type of the domain is inferred.

Introduction of Situation Variables: The plan is re-interpreted in situation calculus and rewritten as nested conditional expression.

The given universal plan represents the optimal transformation sequences for each state of the finite problem domain for which the plan was constructed. To execute a plan, the current input state can be searched in the planning graph by depth-first or breadth-first search starting from the root and the actions along the edges from the current input state to the root can be extracted (Schmid, 1999). If the universal plan is transformed into a finite program, the finite program can be executed by by a functional eval-apply interpreter. For each input state, the resulting transformation sequence should correspond exactly to the sequence of actions associated with that state in the universal plan.

The searched for finite program is already implicitely given in the plan and we have to extract it by plan transformation. A plan can be considered as a finite program for transforming a fixed set of inputs into the desired output by means of applying a total ordered sequence of actions to an initial state, resulting in a state fulfilling the top-level goals. A plan constructed by backward search with the state(s) fulfilling the top-level goals as root, can be read top-down as: *IF the literals at the current node are true in a situation THEN you are done after executing the actions on the path from the current node to the root ELSE go to the child node(s) and recur.*[1] Our goal is to extract the underlying program structure from the plan. To interpret the plan as a program term, states are re-interpreted as boolean operators. All literals of a state description which are involved in transformation (the "footprint", see Veloso, 1994) are rewritten with the predicate symbol as boolean operator introducing a situation variable as additional argument. Additionally, the actions are extended by a situation variable, thus, the current (partial) description of a situation can be passed through the transformations. Finally, additional nodes only containing the situation variable are introduced for all cases, where the current situation fulfills a boolean condition. In this case, the current value of the situation variable is returned.

We define plans as programs in the following way:

Definition 15 (Plan as Program) *Each node S (set of literals) in plan Θ is interpreted as conjunction of boolean expressions B. The planning tree can now be interpreted as nested conditional: IF $B(s)$ THEN t_1 ELSE t_2 with $t_1, t_2 == s \mid o(\Theta(s))$, where s is a situation variable, o the action given at the edge from B to a child node, and Θ' as sub-plan with this child node as root.*

The restriction to binary conditions "if-then-else" is no limitation in expressiveness. Each n-ary condition can be rewritten as nested binary condition:
(cond $(x_1\ t_1)\ (x_2\ t_2)\ (x_3\ t_3) \ldots (x_n\ t_n)) ==$
(if $x_1\ t_1$ *(if* $x_2\ t_2$ *(if* $x_3\ t_3$ *(if* $\ldots$ *(if* $x_n\ t_n\ \Omega)))))$.

If the plan results in a term *IF* $B(s)$ *THEN* s *ELSE* $o(\Theta(s))$, the problem is linear, if *then-* and *else-* part involve operator application, the problem is more complex (resulting in a tree recursion). We will see below, that for

[1] An interpreter function for universal plans is given in (Wysotzki & Schmid, to appear).

some problems a complex structure can be collapsed into a linear one as a result of data type introduction.

For information about the global data structures and central components of the algorithm, see appendix A. In the following we will describe the three steps of plan transformation in an abstract way. The subsequent sections give illustrations for some example domains.

4.1.1 Plan Decomposition

As an initial step the plan might be decomposed in uniform sub-plans:

Definition 16 (Uniform Sub-Plan) *A sub-plan is uniform if it contains only fixed, regular sequences of operator-names* $\langle o_1 \dots o_n \rangle$ *with* $n \geq 1$.

The most simple uniform sub-plan is a sequence of steps where each action involves an identical operator-name (see fig. 4.2.a). It could also be possible, that some operators are applied in a regular way – for example *drill-hole–polish-object* (see fig. 4.2.b). Single operators or regular sequences can alternatively occur in more complex planning structures (see fig. 4.2.c). A plan can contain uniform sub-plans in several ways: The most simple way is, that the plan can be decomposed level-wise (see fig. 4.3.a). In general, sub-plans can occur at any position in the planning structure as subgraphs (see fig. 4.3.b).

We have only implemented a very restricted mode for this initial plan decomposition: single operators and level-wise splitting (see appendix A). A full implementation of decomposition involves complex pattern-matching, which we realize in a later step, when identifying sub-programs in our generalization-to-n approach[2]. Level-wise decomposition can result in a set of "parallel" sub-plans which might be composed again during later planning steps. Parallel sub-plans occur, if the "parent" sub-plan is a tree, i. e., it terminates with more than one leaf. Each leaf becomes the root of a potential subsequent sub-plan.

In the current implementation, we return the complete plan, if different operators occur at the same level. A reasonable minimal extension (avoiding complex pattern-matching) would be, to search for sub-plans fulfilling our simple splitting criterium at lower levels of the plan. But up to now, this case only occurred for complex list problems (as *tower* with 4 blocks, see

[2]Sub-program identification in finite programs and generalization to a set of recursive equations is done by Martin Mühlpfordt in his diploma thesis.

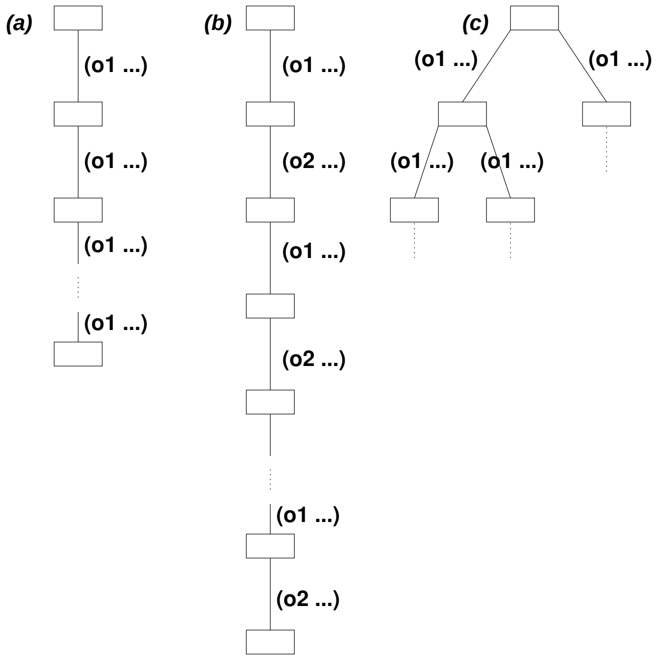


Figure 4.2: Examples of Uniform Sub-Plans

below) and in such cases, a minimal spanning tree is extracted from the plan.

If decomposition results in more than one sub-plan, an initial skeleton for the program structure is generated over the names of the sub-plan, which are initially associated with the partial plans and finally with recursive functions. If generalization-to-n succeeded, the names are extended by the associated lists of parameters. For example, a structure *(p1 (p2 (p3)))* could be completed to *(p1 arg1 (p2 arg2 arg3 (p3 (arg 4 s))))* where the last argument of each sub-program p_i is a situation variable. For all arguments arg_i, the initial values as given in the finite program are known.[3]

4.1.2 Data Type Inference

The central step of plan transformation is data type inference.[4] The structure of a (sub-) plan is used to generate an hypothesis about the underlying

[3]Parameters and initial values are generated in the generalization-to-n algorithm, see Schmid et al., 1999.

[4]Concrete and abstract data types are for example introduced in (Ehrig & Mahr, 1985).

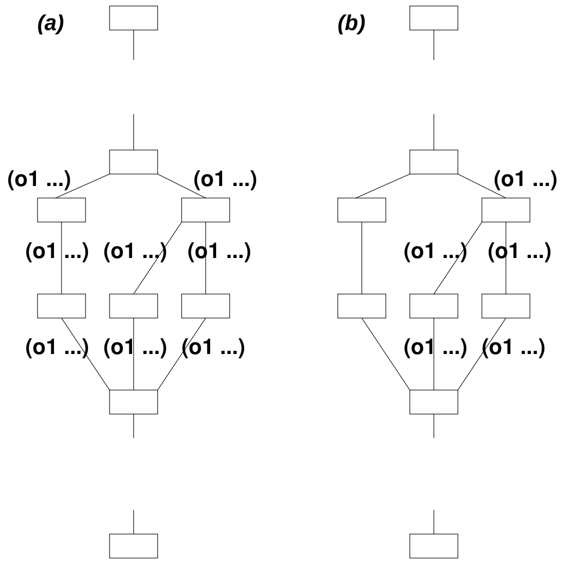


Figure 4.3: Uniform Plans as Subgraphs

data type. This hypothesis invokes certain – data type specific – concepts which subsequently are tried to identify in the plan and certain rewrite-steps which are to be performed on the plan. If the data type specific concepts cannot be identified from the plan, plan transformation fails.

Definition 17 (Data Type) *A data type τ is a collection of data items, with designated basic items $\perp$ (together with a "bottom"-test) and operations (constructors) such that all data items can be generated from basic items by operation-application. The constructor function determines the structure of the data type with $bot < c(x, bot) < c(x', c(x, bot)) \ldots$. If x is empty or unique, the data type is simple and the (countable, infinite) set of elements belonging to τ is totally ordered. The structure of data belonging to complex types is usually a partial order. For complex types, additionally selector functions $el(c(x, s)) = x$ and $rs(c(x, s)) = s$ are defined.*

The data type hypotheses are checked against the plan ordered by increasing complexity:

- Is the plan a sequence of steps (no branching in the plan)?
  Hypothesis: Data Type is *Sequence*

- Does the plan consist of paths with identical sets of actions?
  Hypothesis: Data Type is *Set*

- Is the plan a tree?
  Hypothesis: Data Type is *List* or *compound* type

- Is the plan a DAG?
  Hypothesis: Data Type is *List* or *compound* type.

Data type inference for the different plan structures is discussed in detail below. After the plan is rewritten in accordance to the data type, the order of the operator-applications *and* the order over the domain objects are represented explicitly.

Explicit order over the domain objects is achieved by replacing object names by functional expressions (selector functions) referring to objects in an indirect way. Referring to objects by functions $f(t)$, where t is a ground term (a constant, as the the bottom-element, or a functional expression over a constant) makes it possible to deal with infinite domains while still using finite, compact representations (Geffner, 1999). For example, *(pick oset)* can represent a specific object in an object list of arbitrary length.

4.1.3 Introducing Situation Variables

In the final step of plan transformation, the remaining literals of each state and the actions are extended by situation variable s as additional argument and the plan is rewritten as an conditioned expression as defined in definition 15. An abbreviated version of the rewriting-algorithm is given in appendix A.[5]

4.2 Plans over Sequences of Objects

A plan which consists of a sequence of actions (without branching) is assumed to deal with a sequence of objects. For *sequences*, there has to be a single *bottom* element, which is identifiable from the top-level goal(s) together with the goal-predicate(s) as *bottom-test*. The total order over domain objects is defined over the arguments of the actions from the top (root) of the sequence to the leaf.

[5]This final step is currently only implemented for linearized plans.

Definition 18 (Sequence) *Data type sequence is defined as:*
$seq = \perp \mid c(seq)$ *with*

$$null(seq) = \begin{cases} true & if\ seq = \perp \\ false & otherwise. \end{cases}$$

For a plan – or sub-plan – with hypothesized data type *sequence*, data type introduction works as described in algorithm 5.

Algorithm 5 (Introducing *Sequence*)

- *If the plan starts at level 0:*
  - *If the plan is not a single step:*
    * *If there is a single top-level goal* $(ga_1 \ldots a_n)$ *set* **bottom-test** *to p, else fail.*[6]
    * *Set* **type** *to seq.*
    * *Generate the sequence:*
      Collect the argument-tuple of each action along the path from the root to the leaf.
      If the tuple consists of a single argument, keep it
      otherwise, remove all arguments which are constant over the sequence.
    * *If the sequence consists of single elements and if each element occurs as argument of g,*
      proceed with sequence $= (e_0 \ldots e_m)$ *and set* **bottom** *to* e_0
      else fail.
    * *Construct an association list* $((e_1(succ e_0)) \ldots (e_m(succ^{m-1} e_0)))$.
      For (e_0, e_1) *check, whether the state on level 1 contains a predicate* $q(args)$ *with* $e_0, e_1 \in args$ *at positions* $(pos e_0 1 q)$ *and* $(pos e_1 1 q)$ *if yes, proceed, else fail.*
      For each (e_i, e_{i+1}) *of the sequence with* $i = 1 \ldots m$ *check whether* $q(args)$ *with* $e_i, e_{i+1} \in args$ *exists at level* $i+1$ *with* $(pos e_i, i+1, q) = (pos e_0 1 q) = p_i$ *and* $pos(e_{i+1}, i+1, q) = pos(e_1, 1, q) = p_j$.
      If yes, generate a function $(succ e_i) = e_j\ if$ *(q args) with* e_i *at* p_i *in* q *and* e_j *at* p_j *in* q
      else fail.
    * *Introduce data type* sequence *into the plan:*
      For each state, keep only **bottom-test** *predicate* $(gargs)$
      Replace arguments of g and of actions by $(succ^i e_0)$ *in accordance to the association list.*
  - *If the plan is a single step: identify* bottom-test *and* bottom *as above, reduce states to the bottom-test predicate.*

[6]This step can be extended to multiple goals, if the sequence is generated first. Then, there has to be one remaining constant occurring as argument of the same predicate as the constants involved in the sequence. This predicate has to occur in the set of goal-predicates and is selected as bottom-test predicate. The remaining constant which is argument of this predicate in the goal and which is not involved in rewriting is identified as *bottom*.

- *If the plan starts at a level > 0: an "intermediate" goal has to be identified; afterwards, proceed as above.*

For the application of an inferred recursive control rule, an additional function for identifying successor-elements from the current state has to be provided as described in algorithm 5. The program code for generating this function is given in figure 4.4.

```
; pattern for getting the succ of a constant
; pred has to be replaced by the predicate-name of the rewrite-rule
; x-pos has to be replaced by a list-selector (position of x in pred)
; y-pos dito (position of y = (succ x) in pred)
; sharp-quote #' defines a function
; function-call with (funcall <name> ...)
(setq succ-pattern '(defun succ (x s)
                       (cond ((null s) nil)
                             ((and (equal (first (car s)) pred)
                                   (equal (nth x-pos (car s)) x))
                                  (nth y-pos (car s))
                             )
                             (T (succ x (cdr s)))
                    )))

; use a pattern for calculating the successor of a constant from a
; planning state (succ-pattern) and replace the parameters for the
; current problem
; this function has to be saved so that the synthesized program
; can be executed
(defun transform-to-fct (r)
  ; r: ((pred ...) (y = (succ x)))
  ; pred = (first (car r))
  ; find variable-names x and y and find their positions in (pred ...)
  ; replace pred, x-pos, y-pos
  (setq r-pred (first (car r)))
  (setq r-x-pos (position (second (third (second r))) (first r)))
  (setq r-y-pos (position (first (second r)) (first r)))
  (nsubst (cons 'quote (list r-pred)) 'pred
          (nsubst r-x-pos 'x-pos (nsubst r-y-pos 'y-pos succ-pattern)))
)
```

Figure 4.4: Generating the *Successor*-Function for a Sequence

Unstacking Objects

A prototypical example for plans over a sequence of objects is unstacking objects – either to put all objects on the ground or to clear a specific object located somewhere in the staple. The domain specification and a plan for *unstack* are given in figure 4.5.

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

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

(unstack ?x)
PRE {(clear ?x), (on ?x ?y)}
ADD {(clear ?y)}
DEL {(on ?x ?y)}
(works also for the single PRE {(clear ?x)})

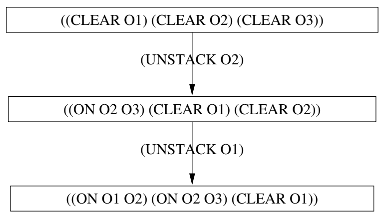


Figure 4.5: The *Unstack* Domain and Plan

The protocol of plan-transformation is given in figure 4.6. After identifying the data type *sequence*, the crucial step is the introduction of the successor-function: *(succ x)* $= y \equiv$ *(on y x)* which represents the "block lying on top of block x". While such functions are usually predefined (Manna & Waldinger, 1987; Geffner, 1999), we can infer them from the universal plan. The "constructively" rewritten plan (data type *sequence* is introduced) is given in figure 4.7, and the finite program in figure 4.8. The transformation information stored for the finite program is given in appendix B.

The finite program written as a term is

```
(IF (CLEAR O3 S)
    S
    (UNSTACK (SUCC O3 S)
        (IF (CLEAR (SUCC O3 S) S)
            S
            (UNSTACK (SUCC (SUCC O3 S) S)
               (IF (CLEAR (SUCC (SUCC O3 S) S) S)
                    S
                    OMEGA))))).
```

An RPS generalizing the *unstack* term is $\Sigma = \langle$ *(unstack-all o s) = (if (clear o s) s (unstack (succ o s) (unstack-all (succ o s) s)))* $\rangle$ with $t =$ *(unstack-rec o s)* for some constant o and some set of literals s. The executable program is given in figure 4.9.

Plans consisting of a linear sequence of operator applications over a sequence of objects in general result in generalized control knowledge in form

```
+++++++++ Transform Plan to Program  ++++++++++
1st step: decompose by operator-type
Single Plan
(SAVE SUBPLAN P1)

(P1)
-----------------------------------------------
2nd step: Identify and introduce data type

(INSPECTING P1) Plan is linear

(SINGLE GOAL-PREDICATE (CLEAR O3))
Plan is of type SEQUENCE

(SEQUENCE IS O3 O2 O1)
(IN CONSTRUCTIVE TERMS THAT 'S (O2 (SUCC O3)) (O1 (SUCC (SUCC O3))))
Building rewrite-cases:
(((ON O2 O3) (O2 = (SUCC O3))) ((ON O1 O2) (O1 = (SUCC O2))))

(GENERALIZED RULE IS (ON |?x1| |?x2|) (|?x1| = (SUCC |?x2|)))
Storage as LISP-function
Reduce states to relevant predicates (footprint)

((CLEAR O3))
((CLEAR (SUCC O3)))
((CLEAR (SUCC (SUCC O3))))
Show Constructive Plan? y
Use GRAPHLET? y
Save graphlet-input to <name>: cplan-unstack
transforming to graphlet syntax... please wait...
transformation ok, now I call graphlet... please wait...
-----------------------------------------------
3rd step: Transform plan to program
Show Plan as Program? y
Save xtree-input to <name>: term-unstack
>>>> You can adjust the window now
Saving this window takes a moment...
Save as gif? y
wrote to term-unstack.gif
---------------------------------------------------------
T
```

Figure 4.6: Protocol for *Unstack*

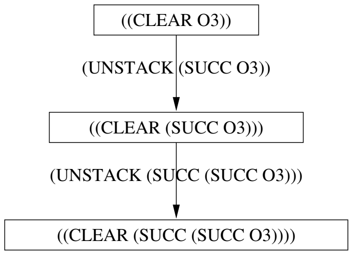


Figure 4.7: Introduction of Data Type *Sequence* in *Unstack*

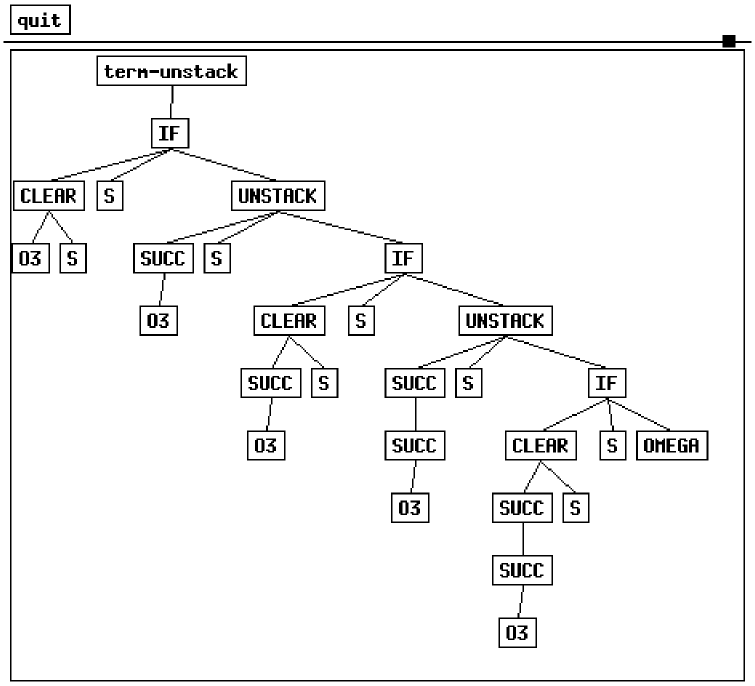


Figure 4.8: Finite Program for the *Unstack* Domain

```
; Complete recursive program for the UNSTACK problem

; call (unstack-all <oname> <state-description>)
; e.g. (unstack-all 'O3 '((on o1 o2) (on o2 o3) (clear o3)))

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

; generalized from finite program generated in plan-transform
(defun unstack-all (o s)
  (if (clear o s)
      s
      (unstack (succ o s) (unstack-all (succ o s) s))
) )

(defun clear (o s)
  (member (list 'clear o) s :test 'equal)
)

; inferred in plan-transform
(DEFUN SUCC (X S)
           (COND ((NULL S) NIL)
                 ((AND (EQUAL (FIRST (CAR S)) 'ON)
                       (EQUAL (NTH 2 (CAR S)) X))
                  (NTH 1 (CAR S)))
                 (T (SUCC X (CDR S)))))

; explicit implementation of "unstack"
; in connection with DPlan: apply unstack-operator on state $s$ and return
; the new state
(defun unstack (o s)
  (cond ((null s) nil)
        ((and (equal (first (car s)) 'on) (equal (second (car s)) o))
                    (cons (cons 'clear (list (third (car s)))) (cdr s))
        )
        (T (cons (car s) (unstack o (cdr s))))
))
```

Figure 4.9: LISP-Program for *Unstack*

Table 4.1: Linear Recursive Functions

(unstack-all x s) ==	(if (clear x) s (unstack (succ x) (unstack-all (succ x) s)))
(factorial x) ==	(if (eq0 x) 1 (mult x (factorial (pred x))))
(sum x) ==	(if (eq0 x) 0 (plus x (sum (pred x))))
(expt m n) ==	(if (eq0 n) 1 (mult m (expt m (pred n))))
(length l) ==	(if (null l) 0 (succ (length (tail l))))
(sumlist l) ==	(if (null l) 0 (plus (head l) (sumlist (tail l))))
(reverse l) ==	(if (null l) nil (append (reverse (tail l)) (list (head l))))
(append l1 l2) ==	(if (null l1) l2 (cons (head l1) (append (tail l1) l2)))

of *linear recursive* functions. In standard programming domains (over numbers, lists), a large group of problems is solvable by functions of this recursion class. Examples are given in table 4.1.

It is simple and straight-forward to generalize over linear plans. For this plans of problems, our learning strategy provides complete and correct control rules. As a result, planning can be avoided completely and the transformation sequence for solving an arbitrary problem involving an arbitrary number of objects can be solved in linear time!

4.3 Plans over Sets of Objects

A plan which has a single root and a single leaf where the set of actions for each path from root to leaf are identical is assumed to deal with a set of objects. For *sets*, there has to be a *complex data object* which is a set of elements (constants of the planning domain), a bottom-element – which is inferred from the elements involved in the top-level goals –, a bottom-test which has to be an inferred predicate over the set, and two selectors – one for an element of a set (*pick*) and one for a set without some fixed element (*rst*). The partial order over sets with maximally three elements is given in figure 4.10 – this order corresponds to the sub-plans for *unload* or *load* in the *rocket* domain. If *pick* and *rst* are defined deterministically (e. g. by list-selectors), the partial order gets reduced to a total order.

Definition 19 (Set) *Data type set is defined as:*
$set = \perp \mid c(e, set)$ *with*

$$empty(set) = \begin{cases} true & if\ set = \perp \\ false & otherwise \end{cases}$$

pick(set) = some $e \in set$
rst(set) = set $\setminus e$ *for some* $e \in set$.

Because the complex data object is inferred from the top-level goals (occurring in the root of the plan), we typically infer an "inverted" order – with the largest set (containing all objects which can be element of set) as bottom element and $c(e, set) == rst(set)$ as a *de*-structor.

For a plan – or sub-plan – with hypothesized data type *set*, data type introduction works as described in algorithm 6. Because we collapse such a plan to one of the set of paths as described in section 2.2.3, this algorithm completely contains the case of sequences (alg. 5). Note, that collapsing plans with underlying data type set corresponds to the idea of "commutative pruning" as discussed in (Haslum & Geffner, 2000).

Algorithm 6 (Introducing *Set*)

- *Collapse plan to one path (implemented as: take the "leftmost" path).*
- *Generate* a complex data object*: like* Generate Sequence *in alg. 5.* *sequence* = $(e1 \ldots e_m)$ *is interpreted as set and* **bottom** *is instantiated with* $CO = (e1 \ldots e_m)$.
  A function for generating CO from the top-level goals (`make-co`) *is provided.*
- *A generalized predicate* (g* args) *with* $CO \in args$ *is constructed by collecting all predicates* (g args) *with* $o \in CO \wedge o \in args$ *and replacing o by CO. For a plan starting at level 0, g has to be a top-level goal and all top-level goals have to be covered by* g^**; for other plans, g has to be in the current root node.*
- *A function for testing whether* g^* *holds in a state is generated as* **bottom-test**.
- *Introduce the data type into the plan:*
  For each state keep only bottom-test predicate (g* args) *with* $CO' \subseteq CO \in args$.
  Introduce set-selectors for arguments of g^* *by replacing* CO' *by* $(rst^i\ CO)$ *and afterwards replacing action-arguments by* $(pick(rst^i\ CO))$ *with* $(rst^i\ CO)$ *occurring as argument of* g^* *of the parent-node.*
  $(pick\ set)$ *and* $(rstset)$ *are predefined by car and cdr.*

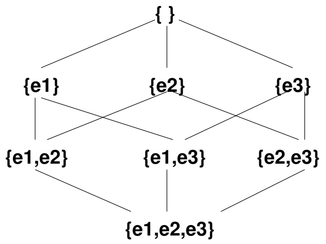


Figure 4.10: Partial Order of *Set*

The program code for `make-co`, generating the bottom-test, and for `pick`, and `rst` referred to in algorithm 6 is given in figure 4.11.

The *rocket* domain as example for a plan over a set of objects was discussed in detail in section 2.2.3. The protocol of plan-transformation is given in figure 4.12.

The recursive functions for loading and unloading all objects in some arbitrary order learned from the four-object *rocket* problem (see sect. 2.2.3) can be of use in many transportation problems, as for example the *logistics* domain[7] which is still one of the most challenging domains for planning algorithms (see the AIPS planning competitions 1998 and 2000).

For the *rocket* domain our system can learn the complete and correct control rules. All problems of this domain can now be solved in linear time. A small flaw is, that generalization-to-n assumes infinite capacity of the transport vehicle and does not take into account capacity as an additional constraint. To get more realistic, we have to include a resource variable[8] for the *load* operator. The resulting *rec-load* function have to involve an additional condition:

```
(load-all oset c s) =
(if (or (eq0 c) (empty oset))
    s
    (load (pick oset) (load-all (rst oset) (prec c) s))).
```

A further extension of the domain would be, to take into account different priorities for objects to be transported. This would involve an extension of *pick*, selecting always the object with highest priority, that is, the control function would follow a *greedy*-algorithm.

A Lisp-program representing the control knowledge for the *rocket* domain is given in appendix B together with a short discussion about interleaving the *inside* and *at* predicates.

[7]The logistics domain defines problems, where objects have to be transported from different places in and between different cities, using trucks within cities and planes between cities.

[8]Dealing with resource-variables is possible with the DPlan-system extended to function applications, as done in the diploma thesis by Marina Müller. Currently we cannot generate disjunctive boolean operators in plan transformation.

```
; make-co
; -------
; collect all objects covered by goal-preds p corresponding to the
; new predicate p*
; the new complex object is referred to as CO
; f.e. for rocket: (at o1 b) (at o2 b) --> CO = (o1 o2)
; how to make the complex object CO: use the pattern of the new
; predicate to collect all objects from the current top-level goal
;; use for call of main function, f.e.: (rocket (make-co ..) s)
;; newpat has to be got from the current MAIN-program!
(defun make-co (goal newpat)
  (cond ((null goal) nil)
        ((string< (string (caar goal)) (string (car newpat)))
             (cons (nth (position 'CO newpat) (car goal))
                   (make-co (cdr goal) newpat)))
))

; rest(set)  (implemented as for lists, otherwise pick/rest would not
; ---------    be guaranteed to be really complements)
;; named rst because rest is build-in
(defun rst (co)
  (cdr co)
)

; pick(set)
(defun pick (co)
  (car co)
)

; is newpred true in the current situation?
;; used for checking this predicate in the generalized function
;; f.e. (at* CO Place s)
;; newpat has to be replaced by the newpred name (f.e. at*)
;; pname has to be replaced by the original pred name (f.e.at)
(setq newpat '(defun newp (args s)
                 (cond ((null args) T)
                       ((and (null s) (not(null args))) nil)
                       ((and (equal (caar s) pname)
                             (intersection args (cdar s)))
                               (newp (set-difference args (cdar s)) (cdr s)))
                       (T (newp args (cdr s)))
              ))
)

(defun make-npfct (patname gpname)
  (subst patname 'newp (nsubst (cons 'quote (list gpname)) 'pname newpat))
)
```

Figure 4.11: Functions inferred/provided for Set

```
+++++++++ Transform Plan to Program  ++++++++++
1st step: decompose by operator-type
Possible Sub-Plan, decompose...
(SAVE SUBPLAN #:P1)
Possible Sub-Plan, decompose...
(SAVE SUBPLAN #:P2)
Single Plan
(SAVE SUBPLAN #:P3)
(#:P1 (#:P2 (#:P3)))
Show Decomposed Plan(s)? n
-----------------------------------------------
2nd step: Identify and introduce data type
(INSPECTING #:P1) Plan is of type SET
Unify equivalent paths...
Introduce complex object (CO)
(CO IS (O1 O2 O3))
A function for generating CO from a goal is provided: make-co
Generalize predicate...
(NEW PREDICATE IS (#:AT* CO B))
Generate a function for testing the new predicate...
New predicate covers top-level goal -> replace goal
Replace basic predicates by new predicate...
Introduce selector functions...
((((O1 O2) (RST (O1 O2 O3))) ((O1) (RST (RST (O1 O2 O3))))
  (NIL (RST (RST (RST (O1 O2 O3))))))
 ((O3 (PICK (O1 O2 O3))) (O2 (PICK (RST (O1 O2 O3))))
  (O1 (PICK (RST (RST (O1 O2 O3)))))))
RST(CO) and PICK(CO) are predefined (as cdr and car).

(INSPECTING #:P2) Plan is linear
Plan consists of a single step
(SET ADD-PRED AS INTERMEDIATE GOAL (AT ROCKET B))

(INSPECTING #:P3) Plan is of type SET
Unify equivalent paths... [... see P1]
Generalize predicate...
(NEW PREDICATE IS (#:INSIDER* CO))
Generate a function for testing the new predicate...
New predicate is set as goal!
Replace basic predicates by new predicate...
Introduce selector functions... [... see P1]
Show Constructive Plan(s)? n
-----------------------------------------------
3rd step: Transform plan to program
Show Plan(s) as Program(s)? n
```

Figure 4.12: Protocol of Transforming the *Rocket* Plan

Table 4.2: Structural Functions over Lists

(a)	(map f l) ==	(if (empty l) nil (cons (f (head l)) (map f (tail l))))
	(inc l) ==	(if (empty l) nil (cons (succ (head l)) (inc (tail l))))
(b)	(reduce f b l) ==	(if (empty l) b (f (head l) (reduce f b (tail l))))
	(sumlist l) ==	(if (null l) 0 (plus (head l) (sumlist (tail l))))
	(rec-unload oset s) ==	(if (empty oset) s
		(unload (pick oset) (rec-unload (rst oset) s)))
(c)	(filter p l) ==	(if (empty l) nil
		(if (p (head l)) (cons (head l) (filter p (tail l))) (filter p (tail l))))
	(odd-els l) ==	(if (empty l) nil
		(if (odd (head l)) (cons (head l) (filter (tail l))) (filter (tail l))))
	(member e l) ==	(if (empty l) nil
		(if (equal (head l) e) e (member e (tail l))))

4.4 Plans over Lists of Objects

4.4.1 Structural and Semantical List Problems

List-problems can be divided in two classes: (a) problems which involve no knowledge about the elements of the list, and (b) problems which involve such knowledge. Standard programming problems of the first class are for example reversing a list, flattening a list, or incrementing elements of a list. Problems, where some operation is performed on every element of a list can be characterized by the higher-order function *(map f l)*. Other list problems which can be solved purely structurally are calculating the length of a list or adding the elements of a list of numbers. Such problems can be characterized by the higher-order function *(reduce f b l)*. The *unload* and *load* problems discussed above fall into this class if each object involved has a unique name and if *pick* and *rst* are realized in a deterministic way. A third class of problems follow *(filter p l)*, for example the functions *member*, or *odd-els*. This class already involves some semantic knowledge about the elements of a list – represented by the predicate p in *filter* and by the *equal* test in *member*. Table 4.2 illustrates structural list-problems.

Structural list problems can be dealt with by an algorithm nearly identical to algorithm 6 dealing with sets. Because the only relevant information is the *length* of a list, the partial order can be reduced to a total order (see fig. 4.13). Generating a total order results in linearizing the problem. The extraction of a unique path in the plan is slightly more complicated as for sets and is discussed below.

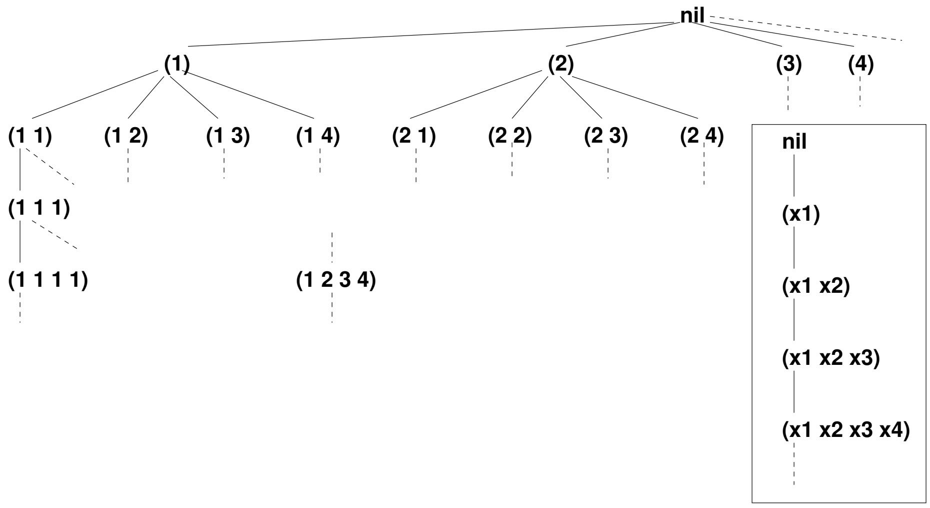


Figure 4.13: Partial Order of *List*

Definition 20 (List) *Data type list is defined as:*
list = nil | cons(e, list) with

$$null(list) = \begin{cases} true & if\ list = nil \\ false & otherwise \end{cases}$$

head(cons(e,list)) = e
tail(cons(e,list)) = list

Algorithm 7 (Introducing *List* (* is discussed in detail below))

- **Collapse plan to one path.*
- *Generate* a complex data object $CO = (e1 \ldots e_m)$.
  A function for generating CO from the top-level goals (`make-co`) *is provided.*
- *A generalized predicate* (g* args) *with* $CO \in args$ *is constructed by collecting all predicates* (g args) *with* $o \in CO \wedge o \in args$ *and replacing o by CO. For a plan starting at level 0 g has to be a top-level goal and all top-level goals have to be covered by g∗; for other plans g has to be in the current root node.*
- *A function for testing whether g∗ holds in a state is generated as* **bottom-test**.
- *Introduce the data type into the plan:*
  For each state keep only bottom-test predicate $(g * args)$ *with* $CO' \subseteq CO \in args$. *Introduce list-selectors by replacing* CO' *by* $(tail^i\ CO)$ *and afterwards replacing action-arguments by* $(head(tail^i\ CO))$ *with* $(tail^i\ CO)$ *occurring as argument of g∗ of the parent-node.*

While functions over lists involving only structural knowledge are easy to infer with our approach – as shown for the *rocket* domain –, this is not true for the second class of problems. A proto-typical example for this class is sorting: for sorting a list, knowledge about which element is smaller (or larger) than another is necessary. That synthesizing functions involving semantic knowledge is notoriously hard is discussed at length in the ILP literature (Flener & Yilmaz, 1999; Le Blanc, 1994).

Currently, we approach transformation for such problems by the steps presented in algorithm 8. We do not claim, that this strategy is applicable to all semantic problems over lists. We developed this strategy from analyzing and implementing plan transformation for the *selection sort* problem, which is described below. We will describe how semantic knowledge can be "detected" by analyzing the structure of a plan. For the future, we plan to investigate further problems and try to find a strategy which covers a class as large as possible.

Algorithm 8 (Dealing with Semantic Information in Lists)

- *Extracting a path (identifying list* structure*):*
  - *Extracting a minimal spanning tree from the DAG: The plan is a DAG, but the structure does not fulfill the set-criterium defined above. Therefore, we cannot just select one path, but we have to extract* one *deterministic set of transformation sequences. For purely structural list-problems every minimal spanning tree is suitable for generalization. For problems involving semantic knowledge only some of the minimal spanning trees can be generalized.*
  - *Regularization of the tree: Generating plan levels with identical actions by shifting nodes downward in the plan and introducing edges with "empty" or "id" actions.*
  - *Collapsing the tree: Unifying identical subtrees which are positioned at the same level of the tree.*
- *If there are still branches left (identifying* semantical *criterium for elements):*
  - *Identify a criterium for classifying elements.*
  - *Unify branches by introducing* list *as argument into operator using the criterium as selection-function.*
- *Proceed using algorithm 7.*

4.4.2 Synthesizing *Selection-Sort*

A Plan for Sorting Lists

The specification for sorting lists with 4 elements is given in table 4.3. In the standard version of DPlan described in this report we only allow for

ADD-DEL-effects and we do not discriminate between static predicates (as *greater than*, being not affected by operator application) and fluid predicates. Note that it is enough to specify the desired position of three of the four list-elements in the goal, because positioning three elements determines the position of the fourth. A more natural specification (for DPlan with functions) is given in table 4.4. This second version allows for plan construction without using a set of predefined states. Information (like which element is on which position in the list or what number is greater than another) can simply be "read" from the list by applying predefined (LISP-) functions. The definition of the *swap*-operator determines whether the problem is solved by *bubble-sort* or by *selection-sort*. In the first case, *swap* is applied to neighbor elements where the first is greater than the other; in the second case, the first condition is omitted. Note, that for finding operator-sequences for sorting a list by an ascending order by *backward* planning, the *greater* condition is reverse!

The universal plan is given in figure 4.14. For sorting a list of 4 elements, there exist 24 states. Swapping elements with the restrictions given for selection sort results in 72 edges. Please note, that we represent plans for sorting abbreviated, writing [1 2 3 4] instead of *((isc p1 1) (isc p2 p) (isc p3 3) (isc p4 4))* and so on. Sorting lists of three elements is illustrated in appendix B.

Different Realizations of *Selection Sort*

To make the transformation steps more intuitive, we first discuss functional variants of *selsort* (see tab. 4.5): The first variant is a standard implementation with two nested *for*-loops. The outer loop processes the list l (more exactly, the array) from start (s) to end e, the inner loop (function *smpos*) searches for the position of the smallest element in l, starting at index $(1+s)$ where s is the current index of the outer loop. The *for*-loops are realized as *tail-recursions*.

There is some conflict between a tail-recursive structure – where some input state is transformed step-wise to the desired output – and a plan – representing a sequence of actions from the *goal* to some initial state. Our definition of a plan as program implies a *linear* recursion (see def. 15). The second variant of selection sort overcomes this discrepancy: For a list l with starting index s and last index e, it is checked, whether l is already sorted from s to a current index c which is initialized with e and step-wise reduced. If yes, the list is returned, otherwise, it is determined which element at positions $(c \ldots e)$ should be swapped to position $(1 - c)$. The

Table 4.3: Specification of *SelSort (isc stands for "(is-content position element)", gt stands for "greater than" and is static)*

$\mathcal{D} = \{$ ((isc p1 1) (isc p2 2) (isc p3 3) (isc p4 4) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 1) (isc p2 2) (isc p3 4) (isc p4 3) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 1) (isc p2 3) (isc p3 2) (isc p4 4) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 1) (isc p2 3) (isc p3 4) (isc p4 2) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 1) (isc p2 4) (isc p3 2) (isc p4 3) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 1) (isc p2 4) (isc p3 3) (isc p4 2) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 2) (isc p2 1) (isc p3 3) (isc p4 4) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 2) (isc p2 1) (isc p3 4) (isc p4 3) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 2) (isc p2 3) (isc p3 1) (isc p4 4) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 2) (isc p2 3) (isc p3 4) (isc p4 1) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 2) (isc p2 4) (isc p3 1) (isc p4 3) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 2) (isc p2 4) (isc p3 3) (isc p4 1) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 3) (isc p2 1) (isc p3 2) (isc p4 4) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 3) (isc p2 1) (isc p3 4) (isc p4 2) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 3) (isc p2 2) (isc p3 1) (isc p4 4) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 3) (isc p2 2) (isc p3 4) (isc p4 1) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 3) (isc p2 4) (isc p3 1) (isc p4 2) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 3) (isc p2 4) (isc p3 2) (isc p4 1) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 4) (isc p2 1) (isc p3 2) (isc p4 3) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 4) (isc p2 1) (isc p3 3) (isc p4 2) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 4) (isc p2 2) (isc p3 1) (isc p4 3) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 4) (isc p2 2) (isc p3 3) (isc p4 1) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 4) (isc p2 3) (isc p3 1) (isc p4 2) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)),
((isc p1 4) (isc p2 3) (isc p3 2) (isc p4 1) (gt 4 3) (gt 4 2) (gt 4 1) (gt 3 2) (gt 3 1) (gt 2 1)) $\}$

$\mathcal{G} = \{$(isc p1 1) (isc p2 2) (isc p3 3)$\}$
$\mathcal{O} = \{$swap$\}$ with

(swap ?p ?q)
PRE {(isc ?p ?n1) (isc ?q ?n2) (gt ?n1 ?n2)}
ADD {(isc ?p ?n2) (isc ?q ?n1)}
DEL {(isc ?p ?n1) (isc ?q ?n2)}

Table 4.4: Specification of *SelSort* Using Functions (*A full description of planning with functions can be found in the diploma thesis of Marina Müller*)

```
(define (domain list-domain)
    (:action swap
              :parameters (?i ?j ?X)
              :precondition ()
              :effect
              ((change (?X in (list ?x) (to (swap ?i ?j ?X)))
              ))
              :post (((list ?X))
                     (> (nth ?j ?X) (nth ?i ?X)))
    ))

(define (problem sort-list)
  :domain    'list-domain
  :variables  ((?i :range (0 2)) (?j :range (0 2)))
  :goal       ((list [ 1 2 3 4]))
)

(defun swap (p1 p2 l)
  (cond ((< p1 p2)
   (substitute (nth p1 l) (nth p2 l)
               (substitute (nth p2 l) (nth p1 l) l :start p1 :end p2)
     :start p2 :end (1+ p2)) )
        (T
   (substitute (nth p2 l) (nth p1 l)
               (substitute (nth p1 l) (nth p2 l) l :start p2 :end p1)
     :start p1 :end (1+ p1)) )
))
```

Table 4.5: Functional Variants for *Selection-Sort*

```
; (1) Standard: Two Tail-Recursions
(defun selsort (l s e)
  (if (= s e)
      l
      (selsort (swap s (smpos s (1+ s) e l) l) (1+ s) e)
))

(defun smpos (s ss e l)
  (if (> ss e)
      s
      (if (> (nth s l) (nth ss l))
          (smpos ss (1+ ss) e l)
          (smpos s (1+ ss) e l)
)))

; (2) Realization as Linear Recursion
; c is ``counter'', starting with last list-position e
(defun lselsort (l c s e)
  (if (sorted l s c)
      l
      (swap* (1- c) c e (lselsort l (1- c) s e))
))

(defun swap* (s from to l)
  (swap s (smpos s from to l) l)
)

(defun sorted (l from c)
  (equal (subseq l from c) (subseq (sort (copy-seq l) '<) from c))
)

; (3) Explicit definition of order (gl is list of pos-key pairs)
; e.g. gl = ((3 4) (2 3) (1 2) (0 1)) --> sorted l = (1 2 3 4)
(defun llselsort (gl l)
  (if (lsorted gl l)
      l
      (lswap* (car gl) (llselsort (cdr gl) l))
))

(defun lsorted (gl l)
  (cond ((null gl) T)
        ((equal (second (car gl)) (car l)) (lsorted (cdr gl) (cdr l)))
        (T nil)
))

(defun lswap* (g l)
  (swap (first g) (position (second g) l) l)
)
```

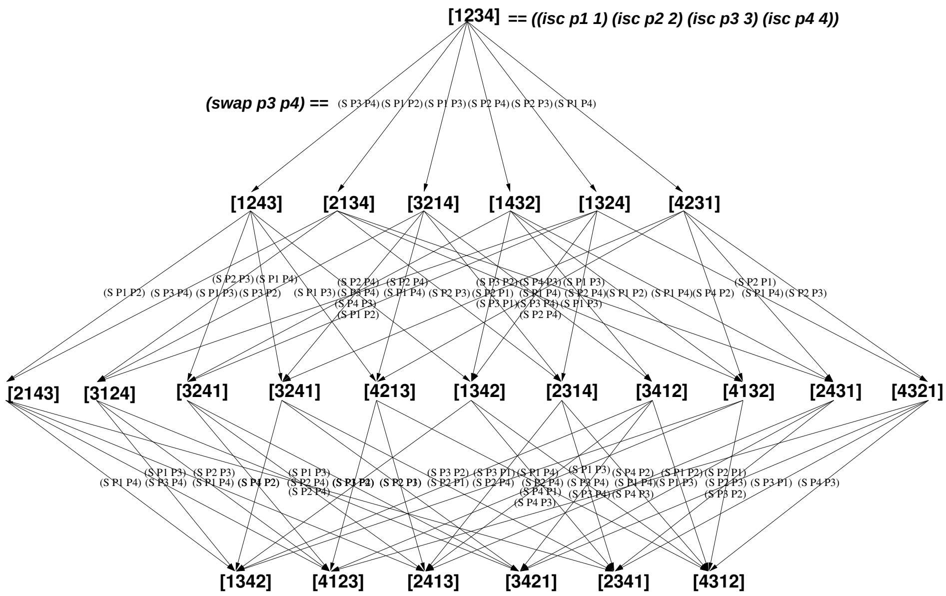


Figure 4.14: Plan for *SelSort (states are represented abbreviated)*

inner loop is replaced by an explicit selector-function. We will see below, that this corresponds to selecting one of several elements represented at different branches on the *same* level of the plan.

The second variant corresponds closely to the function we can infer from the plan, it can be inferred from a plan generated by using function-application.[9] A plan constructed by manipulating literals contains no knowledge about numbers as indices in a list and order relations between numbers. Looking back at plan transformation for *rocket*, we introduced a "complex object" *oset* which guided action application in the recursive *unload* and *load* function. Thus, for *sorting*, we can infer a complex object from the top-level goals which determine which number should be at which position of the list. The third variant gives an abstract representation of the function which we can infer automatically from the plan in figure 4.14. This function is more general than standard selection sort, because now lists can be sorted in accordance to any arbitrary order relation specified in parameter *gl*!

[9] Our investigation of plan transformation for plans involving function-application is still at the beginning.

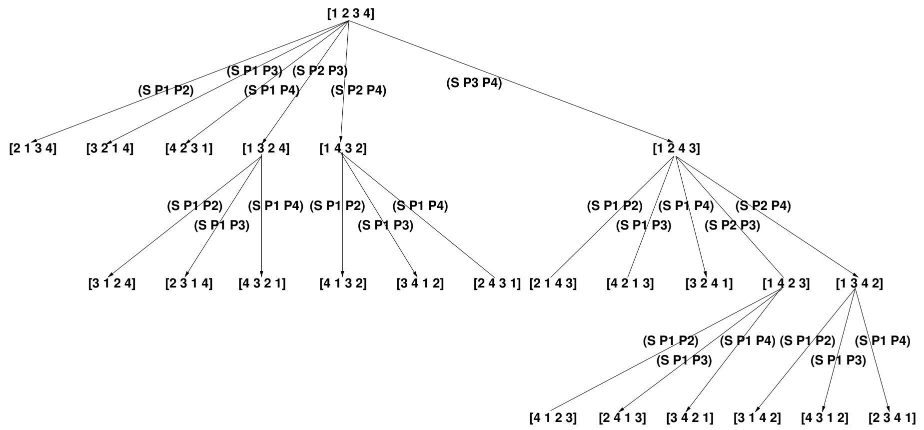


Figure 4.15: A Minimal Spanning Tree Extracted from the *SelSort* Plan

Inferring the Function-Skeleton

The plan given in figure 4.14 is not decomposed by the initial decomposition step, because the complete plan involves only one operator – *swap*. The plan is a DAG, but it does not fulfill the criterium for *sets* of objects. Therefore, extraction of one unique set of optimal plans is done by picking one **minimal spanning tree** from the plan (see sect. 3.3).

The plan for sorting lists with three elements (see appendix B) consists of $3! = 6$ nodes and 9 edges. It contains 9 different minimal spanning trees (see also appendix). Not all of them are suitable for generalization: Three of the nine trees can be "regularized" (see next transformation step below). If we don't have information for picking a suitable minimal spanning tree, we have to extract a tree, try to regularize it and backtrack if regularization fails. For 9 candidates with 3 suitable solutions, this is feasible. But for the sorting of 4 element list, there exist 24 nodes, 72 edges and more than a million possible minimal spanning trees with only a small amount of them being regularizable (see appendix A for calculation of number of minimal spanning trees). Currently, we pre-calculate the number of trees contained in the DAG and if the number exceeds a given threshold t (say 500), we only generate the first t trees. One possible but unsatisfying solution would be to parallelize this step, which is possible. We plan to investigate whether tree-extraction and regularization can be integrated into one step. This would solve the problem in an elegant way. One of the regularizable minimal spanning trees for sorting four elements is given in figure 4.15.

The algorithm for extracting a minimal spanning tree from a DAG is given in alg. 9, the corresponding program fragment in appendix A.

Algorithm 9 (Extract an MST from a DAG)

- *Input: a dag with edges* (nm) *annotated by the level of the node*
- *Initialization:* $t == root(dag)$ (minimal spanning tree)
- *For* $l = 0$ *to* $maxlevel - 1$ *DO:*
  - *Partition all edges* (n m) *from nodes* n *at level* l *to nodes* m *at level* $l + 1$ *into groups with equal end-node* m*:* $p = \{\{(n_1\ m_1)(n_2\ m_1) \ldots (n_k m_1)\}, \ldots \{(n'_1\ m_l) \ldots (n'_{k'}\ m_l)\}\}$.
  - *Calculate the Cartesian product between sets in* p*:* $Cart = p_1 \times p_2 \times \ldots \times p_l$.
  - *Generate trees* $t' = t \cup c$*, for all* $c \in Cart$.

The original plan represented how each of the 24 possible lists over 4 (different) numbers can be transformed into the desired goal (the sorted list) by the set of *all possible* optimal sequences of actions. The minimal spanning tree gives a *unique* sequence of actions for each states. For collapsing a tree into a single path, this tree has to be regular[10]:

Definition 21 (Regular Tree) *An edge-labeled tree* t *with edges* (n m) *from nodes* n *to nodes* m *is regular, if for each level* $l = 0 \ldots maxlevel - 1$ *the subtrees for each node* n $\{(n\ m_1), (n\ m_2), \ldots (n\ m_k)\}$ *consist of id-identical label-sets with* $label \cong label'$ *if* $label = label'$ *or* $label = id$.

The minimal spanning tree is tried to be transformed into a regular tree, using algorithm 10. The program fragment for tree-regularization is given in appendix A. A tree is regularized by pushing all edges labeled with actions occurring also on the next level of the tree down to this level. The starting node of such an edge is "copied" and an *id*-edge is introduced between the original and the copied starting node. Note, that an edge can be shifted more than once. If the result is a regular tree according to definition 21, plan-transformation proceeds, otherwise, it fails.

Algorithm 10 (Regularization of a Tree)

- *Input: an edge-labeled tree*

[10]This definition to regular trees is similar to the definition of redundant attributes in decision trees (Unger & Wysotzki, 1981): If a node has only identical subtrees, the node and all but one subtree are eliminated from the tree.

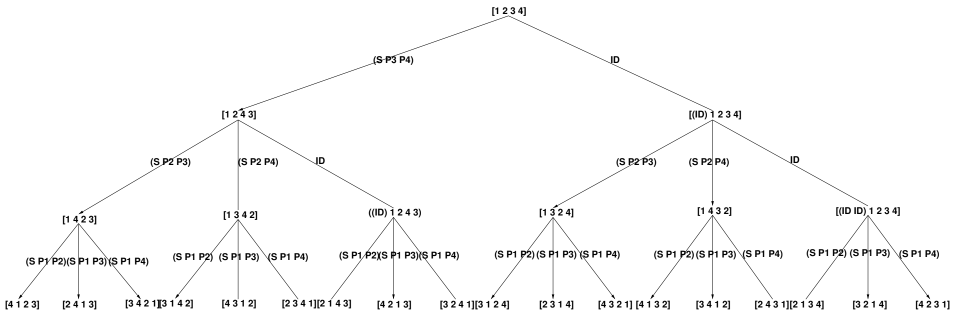


Figure 4.16: The Regularized Tree for *SelSort*

- *For* $l = 0$ *to* $maxlevel - 2$ *DO:*
  - *Construct a label-set* LS_l *for all edges* (n m) *from nodes* n *on level* l *to nodes* m *on level* $l+1$ *and a label-set* LS_{l+1} *for all edges* (o p) *from nodes* o *on level* $l+1$ *to nodes* p *on level* $l+2$.
  - *IF* $LS_{l+1} \subset LS_l$ *shift all edges* $(n_s\ m_s) \in LS_l \cap LS_{l+1}$ *one level and introduce edges* $(n_s\ n_s)$ *with label "id" from level* l *to the shifted nodes* n_s *on level* $n+1$.
- *Test if the resulting tree is regular.*

The regularized version of the minimal spanning tree for *selsort* is given in figure 4.16. The structure of the recursion to be inferred is now already visible in the regularized tree: The "parallel" subtrees have identical edge-labels and the states share a large overlap, as visualized in figure 4.17.

Inferring the "Semantic" Selector Function

Although the *selsort* plan could be reduced successfully to a regular tree with identical subtrees, the plan is still not linear. The remaining subtree as shown in figure 4.17 still contains branches. But this branches again share a regularity:

- ((isc p1 1) (isc p2 2) (isc p3 3)) $\rightarrow$ (swap **p3** p4)
- ((isc p1 1) (isc p2 2)) $\rightarrow$ (swap **p2** {p3,p4})
- ((isc p1 1)) $\rightarrow$ (swap **p1** {p2,p3,p4})

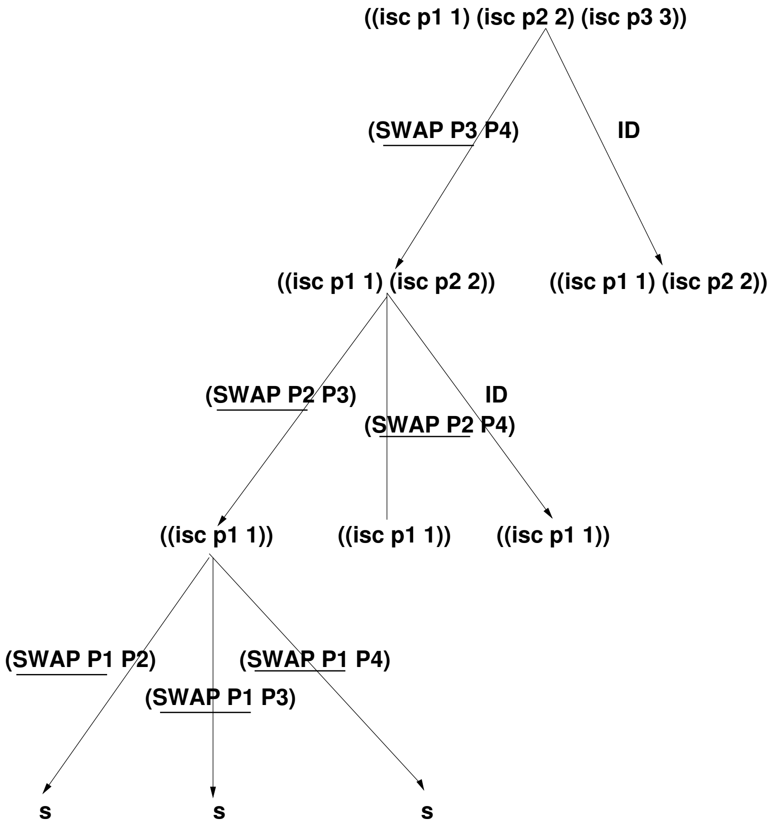


Figure 4.17: The Skeleton of *SelSort* in the Regularized Tree

that is, on each level the first argument of *swap* is constant. From the instantiations of the second argument we can infer the complex object: $CO_S = \{p4\} < \{p3, p4\} < \{p2, p3, p4\}$. Note, that the numbers associated with positions are *not* recognized as numbers. Another minimal spanning tree extracted from the plan, would result in a different pattern, for example $\{p2\} < \{p4, p2\} < \{p1, p4, p2\}$ which is generalizable in the same way as we will describe in the following.

The data object which finally will become the *recursive* parameter is constructed along a *path* in the plan (as described for *rocket*). On each level in the plan, the argument(s) of an action can be characterized relative to the object involved in the *parent* node. Now, we have to introduce a selector function for an argument of an action involving actions on the same level of the plan with the *same* parent node. That is the searched for function has to be defined with respect to the *children* of the action. Remember, that this is a *backward* plan and that a *child*-node is input to an action. As a consequence, the selector function has to be applied to the *current* instantiation of the list (situation).

The searched for function for selecting one element of the candidate elements represented in the second argument of *swap* has to be defined relative to the information available at the current "position" in the plan. That is, the literals of the parent node and the first argument of the *swap* operator can be used to decide which element should be swapped in the current (child) state. For example, for *(swap p3 {p4})*, we have $(sel\ CO_S) = P4$ from *((isc p1 1) (isc p2 2) (isc p3 4) (isc p4 3))*. The first argument of *swap* occurs in *(isc p3 3)* of the parent node. The position to be selected – $p4$ – is related to *(isc p4 3)* in the child node. This observation leads to the hypothesis: For *(swap pos (sel* CO_S*))*, *(sel* CO_S*)* $=$ *pos*$_S$ with *(isc pos key)* $\in$ *parent* and *(isc pos*$_S$ *key)* $\in$ *current*. Because the element which has to be at the position represented by the first argument of *swap* has to be known when *swap* is executed, the hypothesis is extended to:

(swap (pos (isc p3 3)) (sel (key (isc p3 3)) s))*

(sel 3 s) = (pos (find (isc X 3) s))

s = ((isc p1 1) (isc p2 2) (isc p3 4) (isc p4 3))

(pos (isc x y)) = (third (isc x y))

(key (isc x y)) = (second (isc x y)).

Constructing this hypothesis presupposes, that the plan-transformation system has predefined knowledge about how to access elements in lists.[11] Written as Lisp-functions, the general hypothesis is:

```
(defun swap* (fp s)
  (pswap (ppos fp) (sel (pkey fp) s) s) )

(defun sel (k s)
  (ppos (car
  (mapcan #'(lambda(x) (and (equal (pkey x) k) (list x))) s) )))
```

where `pswap` is the *swap* function predefined in the domain specification and `ppos` and `pkey` select the second/third element of a list `(isc x y)`.[12]

The generalized *swap**-function is tested for each *swap*-action occurring in the plan. Because it holds for all cases, plan-transformation can proceed. If the hypothesis would have failed, a new hypothesis had to be constructed. If all hypotheses fail, plan-transformation fails. For the *selsort* plan, only the hypothesis introduced above is possible.

The rest of plan-transformation is straight-forward: The regularized tree is reduced to a single path, unifying branches by replacing the *swap* action by *swap** (see fig. 4.18). Note, that in contrast to the *rocket* problem, this path already contains "id" branches which will constitute the "then" case for the nested conditional to which the plan is finally transformed.

Data type introduction works as described for *rocket* (for *set* and structural *list* problems): a complex object *CO = ((isc p1 1) (isc p2 2) (isc p3 3))* and a generalized predicate *(isc* CO)* for the *bottom*-test is inferred. The state-nodes are rewritten using the rest-selector *tail* on CO. The *head* selector is introduced in the actions: *(swap*(head (tail*^{i} *CO))*. The resulting recursive function is:

```
(defun pselsort (pl s)
  (if (isc* pl s)
```

[11] In the current implementation we provide this knowledge only partially. For example, we can select an element $x \in arg$ from a list *(p arg)*, as needed to construct the definition of *succ* for sequences. That is, we can construct *pos* and *key*. For constructing the definition for *sel*, we need additionally the selection of an element of a list of literals which follows a given pattern. This can be realized with the *filter*-function `mapcan`. But up to know, we did not implement such complex selector functions.

[12] The second condition `(list x)` for `mapcan` is due to the definition of this functor in Lisp. Without this condition, the functor returns `T` or `nil`, with this condition, it returns a list of all elements fulfilling the filter-condition `(equal (pkey x) k)`.

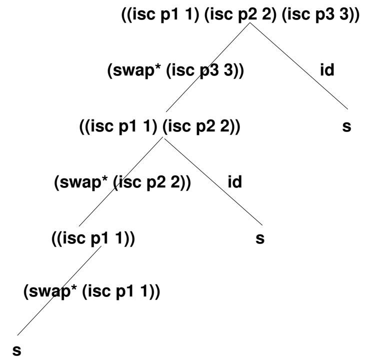


Figure 4.18: Introduction of a "Semantic" Selector Function in the Regularized Tree

```
s
(swap* (head pl)
       (pselsort (tail pl) s)
) ))
```

Note, that *head* and *tail* here correspond to *last* and *butlast*. But, because lists are represented as explicit *position-key* pairs, `pselsort` transforms lists `s` to lists sorted according to the specification derived from the top-level goals given in `pl` independent of the transformation sequence! The Lisp-program realizing selection sort is given in figure 4.19.

Concluding Remarks on List Problems

Transforming the *selsort* plan into a finite program involved to critical steps: (1) extracting a suitable minimal spanning tree from the plan and (2) introducing a "semantic" selector function. The inferred complex object represents the number of elements which are already on the goal position. This is in analogy to the *rocket* problem, where the complex object represented how many objects are already at the goal-destination (at location B for *unload-all* or inside the rocket for *load-all*). Plan transformation results in a final program which can be generalized to a recursive sort function sharing crucial characteristics with *selection sort*. But the function, inferred by our system differs from standard selection sort in two aspects: First, the recursion is *linear*, involving a "goal" stack. The nested for-loops (two *tail*-recursions) of the standard function are realized by a single linear recursive call. Of

```
; Complete Recursive Program for SelSort
; for lists represented as literals
; pl is inferred complex object, e.g. ((p1 1) (p2 2) (p3 3))
; s is situation (statics can be omitted),
;         e.g. ((isc p1 3) (isc p2 1) (isc p3 4) (isc p4 2))

(defun pselsort (pl s)
  (if (isc* pl s)
      s
      (swap* (head pl)
             (pselsort (tail pl) s)
      )))

(defun swap* (fp s)
  (pswap (ppos fp) (sel (pkey fp) s) s) )

(defun sel (k s)
  (spos (car
  (mapcan #'(lambda(x) (and (equal (skey x) k) (list x))) s)
)))

(defun isc* (pl s)
  (subsetp pl (mapcar #'(lambda(x) (cdr x)) s) :test 'equal))

; selectors for elements of pl (p k)
(defun ppos (p) (first p))
(defun pkey (p) (second p))

; selectors for elements of s (isc p k)
(defun spos (p) (second p))
(defun skey (p) (third p))

; head and tail realized as last and butlast
; (from the order defined in the plan, alternatively: car cdr)
(defun head (l) (car (last l)))
(defun tail (l) (butlast l))

; explicit implementation of add-del effect
; in connection with DPlan: application of swap-operator on s
; "inner" union: i=j case
(defun pswap (i j s)
  (print `(swap ,i ,j ,s))
  (let ((ikey (skey (car(remove-if #'(lambda(x) (not (equal i (spos x)))) s))))
        (jkey (skey (car(remove-if #'(lambda(x) (not (equal j (spos x)))) s))))
        (rsts (remove-if #'(lambda(x) (or (equal i (spos x))
                                          (equal j (spos x)))) s))
       )
       (union (union (list (list 'isc i jkey))
                     (list (list 'isc j ikey)) :test 'equal)
              rsts :test 'equal)
))
```

Figure 4.19: LISP-Program for *SelSort*

course, the function for selecting the current position is itself a loop: the literal list is searched for a literal corresponding to a given pattern by an higher-order *filter* function.

We demonstrated plan transformation for a plan for lists with four elements. From a list with three elements, evidence for the hypothesis for generating the semantic selector function would have been weaker (involving only the actions from level 1 to 2 in the regularized tree). An alternative approach to plan transformation, involving knowledge about numbers, is described for a plan for three-element lists in (Wysotzki & Schmid, to appear). In general, there are three backtrack-points for plan transformation:

- Generating "semantic" functions:
  If a generated hypothesis for the semantic function fails or if generalization-to-n fails, generate another hypothesis.

- Extracting a minimal spanning tree from a plan:
  If plan transformation or generalization-to-n fails, select another minimal spanning tree.

- Number of objects involved in planning:
  If plan transformation or generalization-to-n fails, generate a plan, involving an additional object. (A plan has to involve at least three objects which "construct" the recursive parameter for generalization-to-n to succeed, see Schmid et al., 1999.)

Because plan generation has exponential effort (all possible states of a domain for a fixed number of objects have to be generated and the number of states can grow exponentially relative to the number of objects) and because the number of minimal spanning trees might be enormous, generating a finite program suitable for generalization is not efficient in the general case. To reduce backtracking effort, we hope to come up with a good heuristic for extracting a "suitable" minimal spanning tree in the future. One possibility mentioned above is, to try to combine tree extraction and regularization.

4.5 Plans over Complex Data Types

4.5.1 Variants of Complex Finite Programs

The usual way, to classify recursive functions, is to divide them into different complexity classes (Hinman, 1978; Odifreddi, 1989). In complexity theory, the *semantics* of a recursive function is under investigation. For example,

Table 4.6: Structural Complex Recursive Functions

Alternative Tail Recursion	
(max m l) ==	(if (null l) m
	(if (> (head l) m) (max (head l) (tail l)) (max m (tail l))))
Tree Recursion	
(fib x) ==	(if (= 0 x) 0 (if (= 1 x) 1 (plus (fib (- x 1)) (fib (- x 2)))))
μ-Recursion	
(ack x y) ==	(if (= 0 x) (1+ y) (if (= 0 y) (ack (1- x) 1) (ack (1- x) (ack x (1- y)))))

fibonacci is typically implemented as *tree*-recursion (see tab.4.6), but it belongs to the class of linear problems – meaning, the fibonacci-number of a number n can be calculated by a linear recursive function (see Field & Harrison, 1988, pp. 454). In our approach to program synthesis, complexity is determined by the *syntactical structure* of the finite program, based on the structure of a universal plan. The unfolding[13] of all functions in table 4.6 results in a tree structure. Interpretation of *max* always involves only one of the two tail-recursive calls (that is, the function is linear). Interpretation of *fib* results in two new recursive calls for each recursive-step (resulting in an effort $O(2^n)$). The Ackermann-function (*ack*) is the classic example for a not primitive-recursive function with exponential growth – each recursive call results in $y + x$ new recursive calls.

For plan transformation, on the other hand, semantics is taken into account to some extend: As we saw above, plans are linearizable if the data type underlying the plan is a set or a list. For the case of list problems involving semantic attributes of the list elements, it depends on the complexity of the involved "semantic" functions whether the resulting recursion is linear or more complex. Currently, we do not have a theory of "linearizability" of universal plans, but clearly, such a theory is necessary to make our approach to plan transformation more general. A good starting point for investigating this problem, should be the literature on the transformational approach to code optimization in functional programming (Field & Harrison, 1988).

There are two well-known planning domains, for which the underlying data type is more complex than *sets* or *lists*: *Tower of Hanoi* and building a *Tower* of alphabetically sorted blocks in the blocks-world domain. The *tower*

[13]The *unfold* rules for recursive functions are for example presented in (Schmid & Wysotzki, 1998).

Table 4.7: Specification of *Tower* for Three Blocks

$\mathcal{G}$ = {(on A B) (on B C) }
$\mathcal{O}$ = {put, puttable} with

$\mathcal{D}$ = { ((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)) }
(*(ont x)* ("on table") can be used additionally)

(put ?x ?y)
PRE {(ct ?x) (ct ?y)}
$\emptyset \rightarrow$
  ADD {(on ?x ?y)}
  DEL {(ct ?y)}
{(on ?x ?z)} $\rightarrow$
  ADD {(ct ?z)}
  DEL {(on ?x ?z)}

(puttable ?x)
PRE {(ct ?x) (on ?x ?y)[14]}
ADD {(ct ?y)}
DEL {(on ?x ?y)}

problem is a *set of lists* problem and used as one of the benchmark problems for planners. The *hanoi* problem is a *list of lists* problem (the "outer" list is of length 3 for the standard 3 peg problems). For both domains, the general solution procedure to transform an arbitrary state into a state fulfilling the top-level goals is – at least at first glance – more complex than a single linear recursion. Up to now we cannot fully automatically transform plans for such complex domains into finite programs. In the following, we will discuss possible strategies.

4.5.2 The *Tower* Domain

A Plan for Three Blocks

The specification of the three-block *tower* problem is given in table 4.7. The *unstack* domain described above as example for a problem with underlying sequential data type is a *sub-domain* of this problem: the *puttable* operator is structurally identical to the *unstack* operator. The *put* operator has a conditional effect where the ADD-DEL-lists associated with the empty condition are executed in every case.

For the 3-block problem, the universal plan is a unique minimal spanning tree (see fig. 4.20). Note, that for the tower sorted in reverse order to

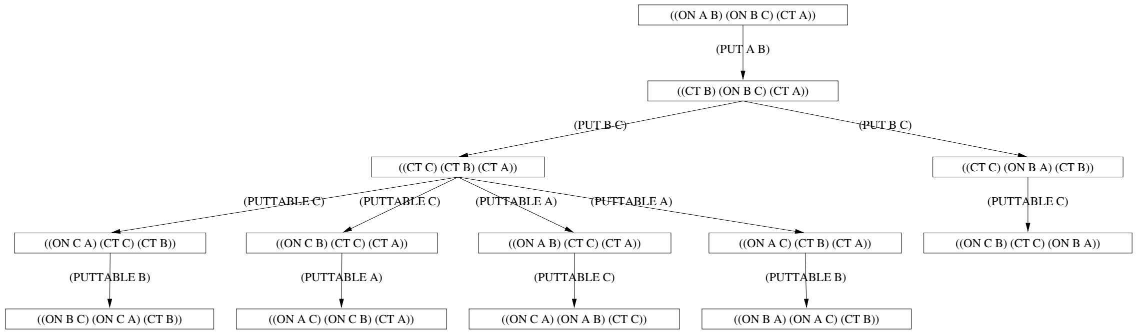


Figure 4.20: Plan for *Tower* with Three Blocks

the goal order of the blocks the transformation sequence is shorter by one action because after the base of the tower C was put on the table, B can immediately put on C without putting it on the table first. Consequently, the control knowledge for solving the *tower* domain should guarantee that *put* is always preferred to *puttable* if already a partial goal tower exists and if the next block for the goal tower is clear.

Assuming Subgoal-Independence

A first strategy for extracting control knowledge for the *tower* domain is described in (Wysotzki & Schmid, to appear): In contrast to the *simultaneous composite learning* strategy described in this report, an *incremental* strategy is used[15]: It is assumed, that the control knowledge for *clearing an arbitrary block* is already available as a *clear*-macro:

$$clear(x,s)=ct(x,s)(s,putt(f(x),clear(f(x),s))).$$

Note, that *clear* immediately returns the current state s, if *ct(x)* already holds in s, i. e., $ct(x) \in s$.

Furthermore, it is assumed, that the subgoals for having a block x on a block y *(on x y)* and for having a block x clear *(ct x)* are *independent*.[16] Therefore, if the top-level goal *(on a b)* is regressed to application of the

[15] In (Wysotzki & Schmid, to appear) additionally to the incremental strategy a simultaneous strategy is discussed.

[16] Note, that this independency assumption corresponds to *linear* planning: It is assumed, that all subgoals of an operator can be fulfilled before the next top-level goal is attacked. Linear planning is incomplete, as was demonstrated for example with the well-investigated Sussman-anomaly (Russell & Norvig, 1995).

put-operator, its preconditions (*(ct a) (ct b)*) are immediately associated with the *clear*-macro, resulting in a linear plan:

$$\text{(on a b) (on b c)} \xrightarrow{(put\ a\ b)\circ(clear\ a)\circ(clear\ b)}$$

$$\text{(on b c)} \xrightarrow{(put\ b\ c)\circ(clear\ b)\circ(clear\ c)} \text{s.}$$

The recursive program which can be inferred presupposing subgoal-independence is given in appendix B. For some domains, it is possible to identify independent predicate sets by analyzing the operator specifications (for example the TIM-analysis for the planner STAN, see Long & Fox, 1999). In contrast, the *puttable*-actions in the universal plan generated using only primitive operators, are generated *after* the *put*-actions! We need the plan for constructing a tower of four blocks, to detect, that *put* and *puttable* can be interleaved[17]!

Simultaneous Composite Learning for *Tower*

Initial plan decomposition for the 3-block *tower* plan results in two sub-plans – a sub-plan for *put-all* and a sub-plan for *puttable-all.* The *put-all* sub-plan is a regular tree as defined above. The only level with branching is for actions *(put B C)* and the plan can be immediately reduced to a linear sequence. Consequently, we introduce the data type *list* with complex object $CO =$ *(A B C)* and bottom-test *(on* (A B C))*. The generalized *put-all* function is structurally analogous to *load-all* from the *rocket* domain:

```
(put-all olist s) ==
  (if (on* olist s)
      s
      (put (first olist) (second olist) (put-all (tail olist) s))
)
```

where `first` and `second` are implemented as `last` and `second-last`, or `olist` gives the desired order of the tower in reverse order.

For the *puttable* sub-plan we have one fragment consisting of a single step – *(puttable C)* – for the reversed tower and a *set* of four sequences:

- $A < C < B$

[17] Note, that we use "interleaving" here in a more general sense than in the planning literature: When we speak of interleaved types of operators, this might not necessarily imply interleaving of goals. During plan construction we used a non-linear technique allowing for goal interleaving.

- $B < C < A$
- $B < A < C$
- $C < A < B$

with *(ct x)* as bottom-test and the constructor *(succ x) = y for (on y x)* as introduced above for linear plans. An obvious strategy compatible with our general approach to plan transformation would be to select one of this sequences and generalize a *clear-all* function identical to the *unstack-all* function discussed above. It remains the problem of selecting the block which is to be unstacked – that is, we have to infer the bottom-element from the goal-set *(ct a) (ct b) (ct c)*. As described for sorting, we have to generate a semantic selector function which is not depended of the parent-node, but of the *current* state.[18]

We have the following examples for constructing the selector function:

((on b c) (on c a)): (sel (A B C)) = A

((on a c) (on c b)), ((on c a) (on a b)): (sel (A B C)) = B

((on b a) (on a c)): (sel (A B C)) = C

that is, for the complex object (ABC) we always have to select the element which is the base of the current tower.

If we model the *tower* domain by explicit use of an *ontable* predicate, this predicate can be used as criterium for the selector function. Without this predicate, we can introduce *(on* CO)* – already generated for *put-all* – and select the *last* element of the list. The resulting *tower* function than would be:

```
(tower olist x) ==
  (put-all olist (clear-all (sel s))
(sel s) == (last (make-olist s)).
```

With the described strategy, the problem got reduced to an underlying data type *list* with a semantic selector function. This control rule generates *correct* transformation sequences for towers with arbitrary numbers of blocks with the desired sorting of blocks specified by `olist`, which is generated from the top-level goals. But, it does not for all cases generate the *optimal* transformation sequences!

[18]Note, that for *selsort* we introduced a selector in the basic operator *swap*. Here we introduce a selector in the function *clear-all* which is already a recursive generalization!

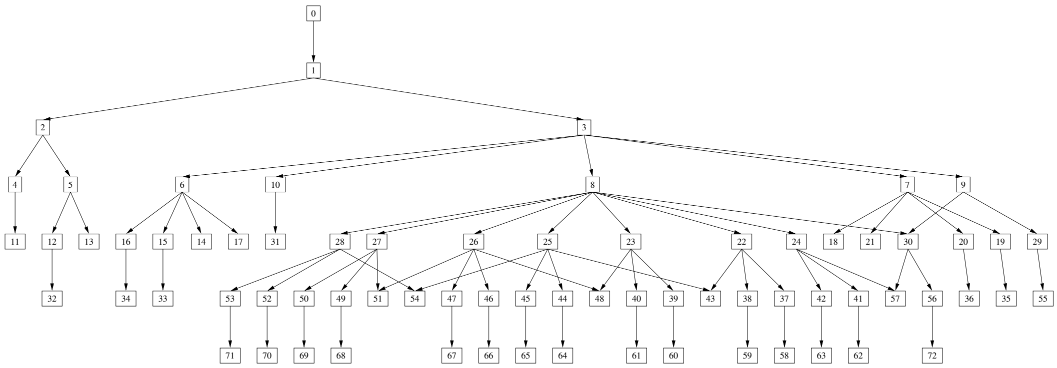


Figure 4.21: Abstract Form of the Universal Plan for the Four-Block *Tower*

We still have not included the single-step case in the *tower* function. For the 3-block plan, we could come up with the discriminating predicate *(on c b)*:

```
(tower olist x) ==
  (put-all olist (if (on c b s)
                     (puttable c s)
                     (clear-all (sel s))
  )                )
```

which generates *incorrect* plans for larger problems, for example for the state *((on c b) (on b a) (on a d) (ct c))*!

For both variants of *tower* a generate-and-test strategy would discover the flaw: For the first variant, it would be detected that for *((on c b) (on b a) (ct a))* an additional action *(puttable b)* would be generated which is not included in the optimal universal plan. For the second variant, all states of the 3-block plan are covered correctly – the faulty condition would only be detected when checking larger problems. But, with only one special case of a reversed tower in the three-block plan every other hypothesis would be highly speculative. Therefore, we now will investigate the four-block plan.

The universal plan for the four-block *tower* problem is a DAG with 73 nodes and 78 edges, thus we have to extract a suitable minimal spanning tree. Because the plan is rather larger, we present an abstract version in figure 4.21 and a summary for the action sequences for all 33 leaf nodes in table 4.8.

For the four-block problem, we have 15 sequences needing to put all blocks on the table and 8 cases with shorter optimal plans (only counting

Table 4.8: Transformation Sequences for Leaf-Nodes of the *Tower* Plan for Four Blocks

15 4-towers, needing 3 `puttable` actions	
((on a b) (on b d) (on d c) (ct a))	(PUTTABLE A) (PUTTABLE B) (PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
((on a c) (on c b) (on b d) (ct a))	(PUTTABLE A) (PUTTABLE C) (PUTTABLE B) (PUT C D) (PUT B C) (PUT A B)
((on a c) (on c d) (on d b) (ct a))	(PUTTABLE A) (PUTTABLE C) (PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
((on a d) (on d b) (on b c) (ct a))	(PUTTABLE A) (PUTTABLE D) (PUTTABLE B) (PUT C D) (PUT B C) (PUT A B)
((on b a) (on a d) (on d c) (ct b))	(PUTTABLE B) (PUTTABLE A) (PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
((on b c) (on c a) (on a d) (ct b))	(PUTTABLE B) (PUTTABLE C) (PUTTABLE A) (PUT C D) (PUT B C) (PUT A B)
((on b c) (on c d) (on d a) (ct b))	(PUTTABLE B) (PUTTABLE C) (PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
((on b d) (on d a) (on a c) (ct b))	(PUTTABLE B) (PUTTABLE D) (PUTTABLE A) (PUT C D) (PUT B C) (PUT A B)
((on c a) (on a b) (on b d) (ct c))	(PUTTABLE C) (PUTTABLE A) (PUTTABLE B) (PUT C D) (PUT B C) (PUT A B)
((on c a) (on a d) (on d b) (ct c))	(PUTTABLE C) (PUTTABLE A) (PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
((on c b) (on b a) (on a d) (ct c))	(PUTTABLE C) (PUTTABLE B) (PUTTABLE A) (PUT C D) (PUT B C) (PUT A B)
((on c b) (on b d) (on d a) (ct c))	(PUTTABLE C) (PUTTABLE B) (PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
((on d a) (on a b) (on b c) (ct d))	(PUTTABLE D) (PUTTABLE A) (PUTTABLE B) (PUT C D) (PUT B C) (PUT A B)
((on d b) (on b a) (on a c) (ct d))	(PUTTABLE D) (PUTTABLE B) (PUTTABLE A) (PUT C D) (PUT B C) (PUT A B)
((on c d) (on d a) (on a b) (ct c))	(PUTTABLE C) (PUTTABLE D) (PUTTABLE A) (PUT C D) (PUT B C) (PUT A B)
	((put c d) (puttable a) also possible)
6 4-towers, needing 2 `puttable` actions	
((on a d) (on d c) (on c b) (ct a))	(PUTTABLE A) (PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
((on b d) (on d c) (on c a) (ct b))	(PUTTABLE B) (PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
((on c d) (on d b) (on b a) (ct c))	(PUTTABLE C) (PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
((on d a) (on a c) (on c b) (ct d))	(PUTTABLE D) (PUTTABLE A) (PUT C D) (PUT B C) (PUT A B)
((on d b) (on b c) (on c a) (ct d))	(PUTTABLE D) (PUTTABLE B) (PUT C D) (PUT B C) (PUT A B)
((on d c) (on c a) (on a b) (ct d))	(PUTTABLE D) (PUT C D) (PUTTABLE A) (PUT B C) (PUT A B)
	((put c d) BEFORE (puttable a)!)
2 4-towers, needing 4 actions	
((on b a) (on a c) (on c d) (ct b))	(PUTTABLE B) (PUTTABLE A) (PUT B C) (PUT A B)
((on d c) (on c b) (on b a) (ct d))	(PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
(sorted tower, 0 actions is root of plan)	
5 2-tower pairs, needing 2 `puttable` actions	
((on a c) (on b d) (ct a) (ct b))	(PUTTABLE B) (PUTTABLE A) (PUT C D) (PUT B C) (PUT A B)
((on a c) (on d b) (ct a) (ct d))	(PUTTABLE D) (PUTTABLE A) (PUT C D) (PUT B C) (PUT A B)
((on a d) (on b c) (ct a) (ct b))	(PUTTABLE A) (PUTTABLE B) (PUT C D) (PUT B C) (PUT A B)
((on b c) (on d a) (ct b) (ct d))	(PUTTABLE D) (PUTTABLE B) (PUT C D) (PUT B C) (PUT A B)
((on a b) (on d c) (ct a) (ct d))	(PUTTABLE D) (PUTTABLE A) (PUT C D) (PUT B C) (PUT A B)
	((put c d) (puttable a) also possible)
5 2-tower pairs, needing 1 `puttable` actions	
((on a d) (on c b) (ct a) (ct c))	(PUTTABLE A) (PUT C D) (PUT B C) (PUT A B)
((on b a) (on d c) (ct b) (ct d))	(PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
((on b d) (on c a) (ct b) (ct c))	(PUTTABLE B) (PUT C D) (PUT B C) (PUT A B)
((on c a) (on d b) (ct c) (ct d))	(PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
((on c b) (on d a) (ct c) (ct d))	(PUTTABLE D) (PUT C D) (PUT B C) (PUT A B)
(2 2-tower pairs, needing no `(put c d)` action	
((on a b) (on c d) (ct a) (ct c))	(PUTTABLE A) (PUT B C) (PUT A B)
((on b a) (on c d) (ct b) (ct c))	(PUT B C) (PUT A B)
are no leafs)	

leaf nodes) – in contrast to 5 to 1 cases for the 3-block tower. Additionally, we have not only one possible partial tower (with 2 or more blocks stacked) but also a two-tower case (with two towers consisting of two blocks). Only one path in the plan makes it necessary to interleave *put* and *puttable*: if D is on top and C immediately under it. There are four cases, where *puttable* has to be performed only two times before the *put* actions are applied and one case, where *puttable* has to be performed only once. For one case, only two *puttable*s and tow *put*s have to be applied. For the case of pairs of towers, all three *put*-actions have to be performed for each leaf, *puttable* has to performed once or twice.

The underlying data type is now not just a list, but a more complicated structure, where for example *(D C A B)* $<$ *D A B C*! Currently, we do not see an easy way to extract all conditions for generating optimal action sequences from the universal plan. Either, we have to be content with the correct but suboptimal control rules inferred from the three-block plan, or we have to rely on incremental learning. A program which covers all conditions for generating optimal plans is given in appendix B.

One path, we want to investigate in the future is, to model the domain specification in a slightly different way – using only a single operator *(put block loc)* where *loc* can be a block or the table. This makes the universal plan more uniform. There is no longer the decision to take, which operator to apply next. Instead, the decision whether a block is put on another block or on the table can be included in the "semantic" selector function.

Set of Lists

Some deeper insight in the structure of the *tower* problem might be gained by analyzing the analogous abstract problem. The sequence of number of states in dependence of the number of blocks is given in table 4.9. This sequence corresponds to the number of sets of lists: *a(n)=(2n-1)a(n-1) - (n-1)(n-2)a(n-2)*. For $n \geq 1$ it is the row sum of the "unsigned Lah-triangle" (see for example, Knuth, 1992). The corresponding formula is $exp(x/(1-x))$.[19]

The *tower* problem is related to generating the power-set of a list with mutually different elements (see table 4.10). But, while for *powerset* different sequences of elements are not included, they have to be *partially* included for *tower*: {*(a) (b c) (b)*} is equal to {*(b) (a) (b c)*} but not to {*(a) (c b) (b)*}. A program generating all different sets of lists (that is *towers*) can be easily

[19] The identification of the sequence was researched by Bernhard Wolf. More background information can be found at `http://www.research.att.com/cgi-bin/access.cgi/as/njas/sequences/eisA.cgi?Anum=000262`.

Table 4.9: Growth of the Number of States for *Tower*

# blocks	1	2	3	4	5
# states	1	3	13	73	501
# blocks	6	7	8	9	10
# states	4051	37633	394353	4596553	58941091
#blocks	11	12	13	14	15
# states	824073141	12470162233	202976401213	3535017524403	65573803186921
#blocks	16	...			
# states	1290434218669921	...			

Table 4.10: Power-set of a List, Set of Lists

```
(defun powerset (l) (pset l (1+ (length l)) (list (list nil))))
(defun pset (l c ps)
  (cond ((= 0 c) nil)
        (T (union ps (pset l (1- c) (ins-el l ps)))) ))
(defun ins-el (l ps)
  (cond ((null l) nil)
        (T (union (mapcar #'(lambda(y) (adjoin (car l) y)) ps)
                  (ins-el (cdr l) ps) :test 'setequal)) ))
                  ; for set of lists  :test 'equal
(defun setequal (s1 s2) (and (subsetp s1 s2) (subsetp s2 s1)))
```

generated from `powerset` by changing `:test 'setequal` to `:test 'equal` in `ins-el`. The *tower* domain is the *inverse* problem to *set of lists*: we want to integrate a set of lists into a single, sorted list using as few operations as possible. The *(puttable x)* operator corresponds to removing an element from a list and generating a new one-element list `(cons (car l) nil)`, the *(put x y)* operator corresponds to removing an element from a list an putting it in front of another list `(cons (car l1) l2)`. A program generating a list of sorted numbers is given in appendix B.

Concluding Remarks on *Tower*

Inference of generalized control knowledge for the *tower* domain was investigated also in the context of two alternative approaches. One of these approaches is genetic programming (Koza, 1992). Within this approach, given primitive operators of some functional programming language together with rules for the correct generation of terms, for a set of input/output examples and an evaluation function (representing knowledge about "good" solutions) a program covering all I/O examples correctly is generated by search in the

Table 4.11: Program for Constructing a *Tower* Generated by Genetic Programming

```
(EQ (DU (MT CS)~(NOT CS)) (DU (MS NN)~(NOT NN)))
```

"equal (
do move-to-table(x) until not(current-stack),
do move-to-stack(next-necessary-block) until not(next-necessary-block)
)"

where next-necessary-block *is identified from a list giving the goal-ordering.*

Table 4.12: Control Rules for *Tower* Inferred by Decision List Learning

A1: PUT-ON $((on_g = on_s) \wedge (\forall on_g^{-1}.holding) \wedge clear_s)$
A2: PUT-ON-TABLE $(holding)$
A3: PICK $((\forall on_g^*.(on_g = on_s)) \wedge (\forall on_g.clear_s) \wedge clear_s)$
A4: PICK $(\neg(on_g^* = on{*}_s) \wedge (\forall on_s.(\forall on_g^{-1}.clear_s)) \wedge clear_s)$
A5: PICK $(\neg(on_g = on_s) \wedge (\forall on *_g .(on_s^* = on_s)) \wedge clear_s)$
A6: PICK $(\neg(on_g = on_s) \wedge (\forall on_s.(\forall on_s^{-1}.clear_s)))$

"evolution space" of programs. The program generated by this approach is given in table 4.11. It corresponds to the "linear" program discussed above. Because always first all blocks are put on the table and afterwards the tower is constructed, the program does not generate optimal transformation sequences for all possible cases.

The second approach, introduced by Martín and Geffner (2000), infers rules from plans for sample input states. The domain is modelled in a concept language (AI knowledge representation language) and the rules are inferred with a decision list learning approach. The resulting rules are given in table 4.12. For example, rule $A3$ represents the knowledge, that a block should be picked up if it's clear, and if its target block is clear and "well-placed". With these rules, 95.5% of 1000 test problems were solved for 5-block problems and 72.2% of 500 test problems were solved for 20-block problems. The generated plans are about two steps longer than the optimal plans. The authors could show, that after a selective extension of the training set by the input states for which the original rules failed to generate a correct plan, a more extensive set of rules is generated for which the generated plans are about one step longer than the optimal plans.

Our approach differs from these two approaches in two aspects: First, we do not use example sets of input/output pairs or of input/plan pairs but we analyze the *complete* space of optimal solutions for a problem of small size. Second, we do not rely on incremental hypothesis-construction, using examples, where the hypothesis fails to guide its modification, but we aim at extracting the control knowledge from the given universal plan by exploiting the structural information contained in it. Although we failed up to now to generate optimal rules for *tower*, we could show for *sequence*, *set*, and *list* problems, that with our analytical approach we can extract correct and optimal rules from the plan.

There is a trade-off between optimality of the policy versus (a) the efficiency of control knowledge application and (b) the efficiency of control knowledge learning. As we can see from the program presented in appendix B and from the (*still non-optimal!*) control rules in table 4.12, generating minimal action sequences might involve complex tests which have to be performed on the current state. In the worst case, these tests again involve recursion, for example, a test, whether already a "well-placed" partial tower exists (test `subtow` in our program). Furthermore, we demonstrated, that the suboptimal control rules for *tower* could be extracted quite easily from the 3-block plan, while automatic extraction of the optimal rules from the 4-block plan involves complex reasoning (for generating the tests for "special" cases).

4.5.3 Tower of Hanoi

Up to now, we did not investigate plan transformation for the Tower of Hanoi. Thus, we will make just some more general remarks about this domain. It is often claimed, that *hanoi* is a highly artificial domain, and that the only isomorphic domains are hand-crafted puzzles, as for example the *monster* problems (Simon & Hayes, 1976; Clément & Richard, 1997). I want to point out, that there are *solitaire* ("patience") games, which are isomorphic to *hanoi*.[20] One of these solitaire-games (freecell) was included in the AIPS-2000 planning competition.

The domain specification for *hanoi* is given in table 4.13. The resulting plan is a unique minimal spanning tree, which is already regular (see figure 4.22). This indicates, that data type inference and resulting plan transformation should be easier than for the *tower* problem. While *hanoi* with *three* discs contains more states than the three-block *tower* domain (27 to 13) the actions for transforming one state into another are much more restricted.

[20]We plan to conduct a psychological experiment in the domain of problem solving by analogy, demonstrating, that subjects who are acquainted with playing patience games perform better on *hanoi* than subjects with no such experience.

Table 4.13: Specification for *Hanoi*

States are represented by the literals *(on disc loc)* and *(ct loc)* where *loc* is either a disc or a peg. Furthermore, static literals *(smaller loc disc)* are used, which are necessary to constrain the *move* operator to legal moves.

$\mathcal{G} = \{$(on d3 p3) (on d2 d3) (on d1 d2) $\}$

$\mathcal{O} = \{$move$\}$ with

(move ?d ?from ?to)
PRE {(on ?d ?from) (ct ?d) (smaller ?to ?d) (ct ?to)}
ADD {(on ?d ?to) (ct ?from)}
DEL {(on ?d ?from) (ct ?to)}

The number of states for *hanoi* is 3^n. The minimal number of moves when starting with a complete tower on one peg is 2^{n-1}. Up to now, there seems to be no general formula to calculate the minimal number of moves for an *arbitrary* starting state – that is, one of the nodes of the universal plan.[21]

Tower of Hanoi is a puzzle investigated extensively in artificial intelligence as well as in cognitive psychology since the 60ies. In computer science classes, Tower of Hanoi is used as a prototypical example for a problem with exponential effort. Coming up with efficient algorithms (for restricted variants) of the Tower of Hanoi problem is still ongoing research (Atkinson, 1981; Pettorossi, 1984; Walsh, 1983; Allouche, 1994; Hinz, 1996). As far as we survey the literature, all algorithms are concerned with the case, where a tower of n discs is initially located at a predefined start peg (see for example table 4.14). In general, *hanoi* is μ-recursive already for the restricted state where the initial state is fixed and only the number of discs are variable with the structure *hanoi* $\circ$ *move* $\circ$ *hanoi*. A standard implementation, as shown in table 4.14 is as tree-recursion.

We are interested in *learning* a control strategy starting with an *arbitrary initial state* (see program in table 4.15).

[21] see: `http://forum.swarthmore.edu/epigone/geometry-puzzles/twimclehmeh/7oen0r212cwy@forum.swarthmore.edu`, open question from Februar 2000

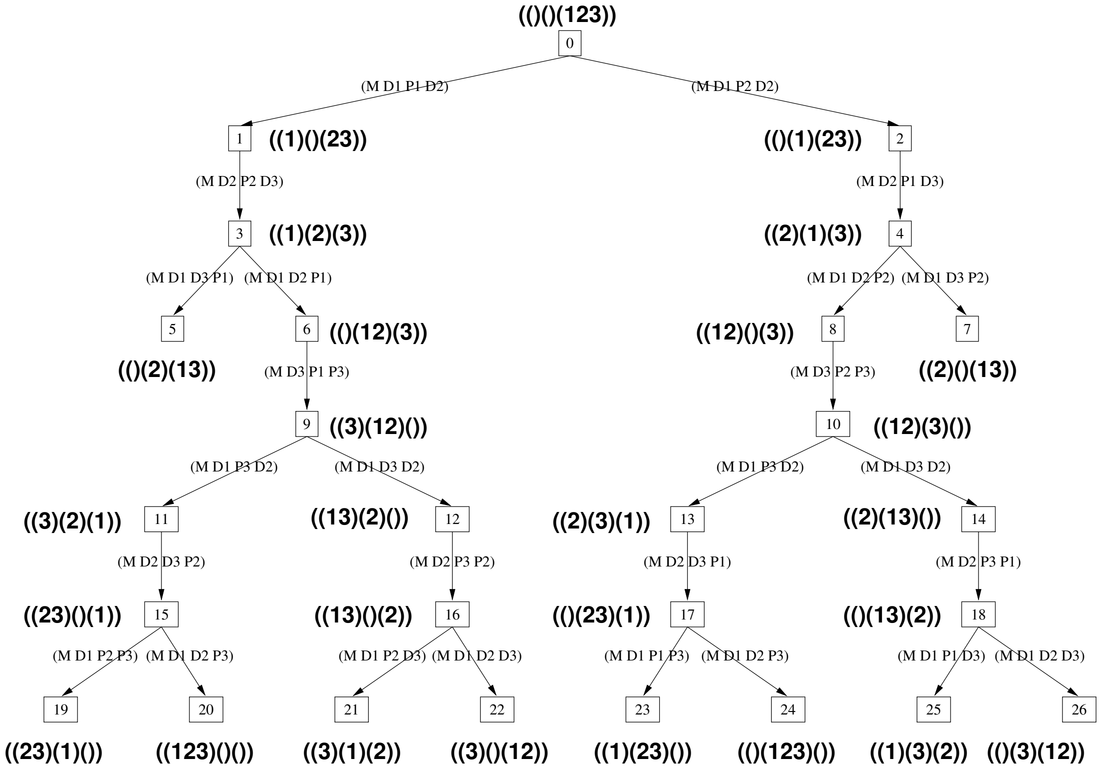


Figure 4.22: Plan for *Hanoi*

Table 4.14: A Tower of Hanoi Program

```
; (SETQ A '(1 2 3) B NIL C NIL) (start) OR
; (hanoi '(1 2 3) nil nil 3)

(DEFUN move (from to)
    (COND ( (NULL (EVAL from)) (PRINT (LIST 'PEG from 'EMPTY)) )
          ( (OR (NULL (EVAL to))
                (> (CAR (EVAL to)) (CAR (EVAL from)) ))
            (SET to (CONS (CAR(EVAL from)) (EVAL to)) )
            (SET from  (CDR (EVAL from))                  )
          )
          ( T  (PRINT (LIST 'MOVE 'FROM (CAR(EVAL from))
                                      'TO (CAR(EVAL to)) 'NOT 'POSSIBLE)))
    )
  (LIST(LIST 'MOVE 'DISC (CAR (EVAL to)) 'FROM  from 'TO to)) )

(DEFUN hanoi (from to help n)
   (COND ( (= n 1) (move from to) )
         ( T  ( APPEND
                  (hanoi from help to (- n 1))
                  (move from to)
                  (hanoi help to from (- n 1))
) )      )     )

(DEFUN start () (hanoi  'A 'B 'C (LENGTH A)))
```

Table 4.15: A Tower of Hanoi Program for Arbitrary Starting Constellations

```
(DEFUN ison (disc peg)
  (COND ( (NULL peg) NIL)
        ( (= (CAR peg) disc) T)
        ( T  (ison disc (cdr peg)))))
(DEFUN on (disc from to help)
 (COND ( (ison disc (eval from))  from)
       ( (ison disc (eval to)) to)
       ( T                       help)))
; whichpeg: peg on which the current disc is NOT lying and peg which
; is not current goal peg
(DEFUN whichpeg (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) ))
(DEFUN topof (peg)
 (COND ( (null (car (eval peg))) nil) ( T   (car  (eval peg))) ))
(DEFUN clearpeg (peg)
 (COND ( (null (car (eval peg))) T) ( T nil) ))
(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)))

(DEFUN gmove (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 'MOVE 'FROM disc 'ON peg 'NOT 'POSSIBLE)))))

(DEFUN ghanoi (disc peg)
 (COND ( (and (= disc 1) (equal (on disc 'A 'B 'C) peg)) T )
 ( T (COND
       ( (equal (on disc 'A 'B 'C) peg) (ghanoi (- disc 1) peg) )
       ( (and (not (equal (on disc 'A 'B 'C) peg))
              (not (and (cleartop disc) (clearpeg peg)))
              (> disc 1)) (ghanoi (- disc 1) (whichpeg disc peg)) )
    )
   (gmove disc peg)
   (COND ((> disc 1) (ghanoi (- disc 1) peg))) )))

(DEFUN n-of-discs (p1 p2 p3) (+ (LENGTH p1) (+ (LENGTH p2) (LENGTH p3))))
; ghanoi: "largest" Disc x Goal-Peg --> Solution Sequence
(DEFUN gstart () (ghanoi (n-of-discs A B C) 'C))
```

Chapter 5

Conclusions and Further Work

We reported work in progress in the context of a larger project on combining planning and inductive program synthesis. The focus of this paper was on transformation of plans into finite programs which can be folded into recursive functions by a generalization-to-n technique. Our approach relies on exploiting the structural information given in a plan. In contrast to techniques where generalized rules are constructed incrementally over problem solving experience (Veloso et al., 1995) or from training examples (Koza, 1992; Martín & Geffner, 2000), our starting point is a universal plan representing the complete knowledge for transforming the complete set of states from a small problem space into a state fulfilling the top-level goals.

We introduced data type inference as crucial step for plan transformation: a universal plan already represents the control structure of the searched for program explicitly containing the shortest transformation sequences for each input into the desired output. For constructing a program term, additional knowledge about the data type which is manipulated by the program is needed – that is, the order of the elements belonging to this type together with selector-functions for accessing components of complex data types. We demonstrated, that information about the data structure underlying a problem domain is already contained in the universal plan and can be extracted by analyzing the structure of the plan. For problems which are solvable by purely structural manipulations, data type extraction is sufficient for generating a program term. For example, for unstacking some block in a stack of blocks, knowledge about the blocks lying on top of that block (as color or weight) is not necessary. For problems, where semantic characteristics of the involved objects determine the transformation sequence, preliminary to data type inference a "semantic" selector function has to be inferred from the plan. This selector function determines which subset of a set of elements can be used as argument to an operator in a given context! For example, when sorting a list of elements using a *swap*-operator, the element to be swapped to the current position can always be the one with the smallest value contained in the list right of the current position (as described for *selsort* in the last chapter).

Transportation domains as *rocket*, have a natural extension to taking into account characteristics of objects: if there are more objects than can be loaded into the rocket, these objects might be selected by some criteria (as value or weight; leading to a greedy-algorithm). While data type inference, introducing the data type into the plan and rewriting the plan into a finite program are fully implemented, we are still at the beginning of formalizing and implementing the generation of semantic selector functions. A possible approach might be the use of decision trees to extract the relevant (possibly context dependent) attributes to classify data objects as positive or negative candidates for operator-arguments.

We discussed efficiency concerns throughout the paper, but we did not perform empirical studies to compare planning effort with application of domain specific control rules. If we can extract generalized control rules from a plan, search can be omitted completely for all possible problems of this domain and the effort of generating the optimal solution sequence corresponds to the effort of executing the recursive function(s) – e. g., $2n+1$, i. e. a linear effort, for the *rocket* problem with n objects, in contrast to a worst case effort of n^2 (loading each of the objects alone and driving to the destination, loading pairs of objects, ...) for plan generation.

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

Our work contributes to planning research, providing a learning technique to make domain-independent planning more efficient without making domain modelling more sophisticated. After the rise of a generation of more efficient planning strategies in the 90ies (Blum & Furst, 1997; Koehler et al., 1997; Kautz & Selman, 1998), inference of domain specific control knowledge to scale-up planning further, is one of the rising research topics (Martín & Geffner, 2000; Long & Fox, 2000, 1999; McCluskey & Porteous, 1997).

Furthermore, we propose that our work provides some useful inside for knowledge based software engineering: One of the most successful systems which assist program development for complex domains is KIDS (Smith, 1990). The system gains its power from cleverly hand-crafted program schemes – like *divide-and-conquer* – which are refined in accordance to a current specification. The schemes are provided by human experts which have a deep understanding of program structures and a long experience in program development. Our learning approach can be viewed as a model for how such schemes can be extracted from experience. If we have more insight into the cognitive processes responsible for generating expertise, this knowledge can be used to make expertise more available and more transparent to a larger group of program developers in teaching or in interactive support systems.

Finally, we believe, that our works contributes to the research on cognitive models of skill acquisition. Here, learning is generally modelled by "chunking" already predefined rules (see the ACT production system architecture, Anderson & Lebiere, 1998) – similar to the early work on linear macros in planning (Minton, 1985). As a consequence, in a production system as ACT, for more complex problems rules interact over a stack of open goals, while the strategy for the sequence of rule applications is "hidden" in the interpreter. Transforming this approach to human problem solving, it has to be assumed that a human keeps track of the still open goals by storing a goal stack in working memory! If, on the other hand, humans are able to extract the generalized rule or solution scheme from some problem solving experience, as it is for example discussed for the Tower of Hanoi, they have the control knowledge for deciding what operator to apply when explicitly available and can use it to generate the solution sequence. Current empirical work in the domain of high-school algebra shows that students do better, if they infer a generalized solution scheme from a small example rather than when they are presented with the scheme first (Koedinger & Anderson, 1998). Our program synthesis technique demonstrates how a generalized strategy can be learned from experience. It addresses one of the original questions of the early cognitive psychology – how rules can be extracted from unstructured perception (c. f., the *language acquisition devise*, Chomsky, 1959). A second aspect of human learning and expertise for which no adequate models exist in cognitive psychology is the development of perceptual chunks (Koedinger & Anderson, 1990). We propose, that data type inference addresses this question: For example, for the *rocket* problem, the *size of the set* (and not some fixed sequence) of objects which are at the starting location is the relevant information unit for estimating how many steps are needed until the rocket can move to its destination. We do not claim that our approach models cognitive processes adequately or in any way similar to human information processes, but we believe that it provides a useful analytical framework – describing what initial information is needed to and what algorithmic strategies are necessary to automatically infer perceptive chunks and problem solving strategies.

References

Allouche, J.-P. (1994). Note on the cyclic towers of hanoi. *Theoret. Comput. Sci.*, *123*, 3-7.

Anderson, J. R., & Lebiere, C. (1998). *The atomic components of thought.* Lawrence Erlbaum.

Atkinson, M. D. (1981). The cyclic towers of hanoi. *Information Processing Letters*, *13*(3), 118-119.

Blum, A., & Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, *90*(1-2), 281-300.

Bonet, B., & Geffner, H. (1999). Planning as heuristic search: New results. In *Proc. European Conference on Planning (ECP-99), Durham, UK.* Springer.

Borrajo, D., & Veloso, M. (1996). Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning*, *10*, 1–34.

Briesemeister, L., Scheffer, T., & Wysotzki, F. (1996). A concept based algorithmic model for skill acquisition. In U. Schmid, J. Krems, & F. Wysotzki (Eds.), *Proceedings of the First European Workshop on Cognitive Modeling (14.-16.11.96).* TU Berlin.

Chomsky, N. (1959). Review of skinner's 'verbal behavior'. *Language*, *35*, 26-58.

Christofides, N. (1975). *Graph theory - an algorithmic approach.* London: Academic Press.

Cimatti, A., Roveri, M., & Traverso, P. (1998). Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)* (pp. 875–881). Menlo Park: AAAI Press.

Clément, E., & Richard, J.-F. (1997). Knowledge of domain effects in problem representation: The case of Tower of Hanoi isomorphs. *Thinking and Reasoning*, *3*(2), 133-157.

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

Courcelle, B., & Nivat, M. (1978). The algebraic semantics of recursive program schemes. In Winkowski (Ed.), *Math. foundations of computer science* (Vol. 64, p. 16-30). Springer.

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

Ehrig, H., & Mahr, B. (1985). *Fundamentals of algebraic specification 1 - equations and initial semantics.* Berlin: Springer.

Field, A., & Harrison, P. (1988). *Functional progamming.* Reading, MA: Addison-Wesley.

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

Geffner, H. (1999). *Functional strips: a more flexible language for planning and problem solving.* (Submitted. Earlier version presented at "Logic-based AI Workshop", Washington D.C., June 1999)

Haslum, P., & Geffner, H. (2000). Admissible heuristics for optimal planning. In *Proceedings of the AIPS 2000* (p. 150-158). AAAI Press.

Hinman, P. (1978). *Recursion-theoretic hierarchies.* New York: Springer.

Hinz, A. M. (1996). Square-free tower of hanoi sequences. *Enseign. Math., 2*(42), 257-264.

Jensen, R. M., & Veloso, M. M. (in press). OBDD-based universal planning: Specifying and solving planning problems for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research.*

Kalmar, Z., & Szepesvari, C. (1999). *An evaluation criterion for macro learning and some results* (Tech. Rep. No. TR99-01). Mindmaker Ltd., Budapest 1121, Konkoly Th. M. u. 29-33.

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

Kautz, H., & Selman, B. (1998). Blackbox: A new approach to the application of theorem proving to problem solving. In *AIPS98 Workshop on Planning and Combinatorial Search* (p. 58-60). Pittsburgh, PA.

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

Knuth, D. E. (1992). Convolution polynomials. *The Mathematica J., 2*, 67-78.

Koedinger, K., & Anderson, J. (1990). Abstract planning and perceptual chunks: Elements of expertise in geometry. *Cognitive Science, 14*, 511-550.

Koedinger, K., & Anderson, J. (1998). Illustrating principled design: The early evolution of a cognitive tutor for algebra symbolization. *Interactive Learning Environments, 5*, 161-180.

Koehler, J. (1998). Planning under resource constraints. In H. Prade (Ed.), *13th European Conference on Artificial Intelligence.* Wiley.

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

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

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

Le Blanc, G. (1994). BMWk revisited: Generalization and formalization of an algorithm for detecting recursive relations in term sequences. In F. Bergadano & L. de Raedt (Eds.), *Machine Learning, Proc. of ECML-94* (p. 183-197). Springer.

Long, D., & Fox, M. (1999). The efficient implementation of the plan-graph in STAN. *Journal of Artificial Intelligence Research*, *10.*

Long, D., & Fox, M. (2000). Automatic synthesis and use of generic types in planning. In *Proceedings of the AIPS 2000* (p. 196-205). AAAI Press.

Mädler, F. (1992). Problemzerlegung als optimalitätserhaltende Operatorabstraktion. *KI*, *2*, 37-41.

Manna, Z., & Waldinger, R. (1975). Knowledge and reasoning in program synthesis. *Artificial Intelligence*, *6*, 175-208.

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

Martín, M., & Geffner, H. (2000). Learning generalized policies in planning using concept languages. In *Proc. 7th Int. Conf. on Knowledge Representation and Reasoning (KR 2000, Colorado, 4/2000).* Morgan Kaufmann.

McCluskey, T., & Porteous, J. (1997). Engeneering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence*, *95*, 1-65.

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

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

Müller, M. (2000). *Integration von Funktionsanwendungen beim zustandsbasierten Planen.* diploma thesis, Dep. of Computer Science, TU Berlin.

Odifreddi, P. (1989). *Classical recursion theory* (Vol. 125). Amsterdam: North-Holland.

Parandian, B., Schmid, U., & Wysotzki, F. (1995). Program synthesis with a generalized planning approach. In *Program synthesis by learning and planning.* Technical Report, Department of Computer Science, TU Berlin.

Peot, M., & Smith, D. (1992). Conditional nonlinear planning. In *1st*

International Conference on AI Planning Systems, AIPS-92 (pp. 189–197). Morgan Kaufmann.

Pettorossi, A. (1984). *Towers of hanoi problems: Deriving the iterative solutions using the program transformation technique* (Tech. Rep. No. R.82). Roma: Instituto di Analisi dei Sistemi ed Informatica.

Precup, D., & Sutton, R. (1998). Multi-time models for temporally abstract planning. In *Advances in neural information processing systems* (Vol. 10). Cambridge, MA: MIT Press.

Russell, S. J., & Norvig, P. (1995). *Artificial intelligence. A modern approach.* Englewood Cliffs, NJ: Prentice-Hall.

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

Schmid, U. (1999). *Iterative macro-operators revisited: Applying program synthesis to learning in planning* (Tech. Rep. No. CMU-CS-99-114). Computer Science Department, Carnegie Mellon University.

Schmid, U., Brandes, O., & Wysotzki, F. (1997). *Simultaneous planning for sets of states.* (unpublished draft, 12 pages)

Schmid, U., Mühlpfordt, M., & Wysotzki, F. (1999). *Induction of recursive program schemes as inference of context free tree grammers.* (draft)

Schmid, U., & Wysotzki, F. (1998). Induction of recursive program schemes. In *Proceedings of the 10th European Conference on Machine Learning (ECML-98)* (p. 214-225). Springer.

Schmid, U., & Wysotzki, F. (2000). Applying inductive programm synthesis to macro learning. In *Proceedings of the AIPS 2000* (p. 371-378). AAAI Press.

Schoppers, M. (1987). Universal plans for reactive robots in unpredictable environments. In *IJCAI '87* (p. 1039-1046). Morgan Kaufmann.

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

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

Simon, H. A., & Hayes, J. R. (1976). The understanding process: Problem isomorphs. *Cognitive Psychology*, *8*, 165-190.

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

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

Sun, R., & Sessions, C. (1999). *Self-segmentation of sequences: Automatic formation of hierarchies of sequential behaviors* (Tech. Rep. No. 609-

951-2781). NEC Research Institute, Princeton.

Unger, S., & Wysotzki, F. (1981). *Lernfähige Klassifizierungssysteme.* Berlin: Akademie-Verlag.

Veloso, M. (1994). *Planning and learning by analogical reasoning.* Springer.

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

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

Walsh, T. (1983). Iteration strikes back - at the cyclic towers of hanoi. *Information Processing Letters*, *16*, 91-93.

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

Wysotzki, F. (1987). Program synthesis by hierarchical planning. In P. Jorrand & V. Sgurev (Eds.), *Artificial intelligence: Methodology, systems, applications* (p. 3-11). Amsterdam: Elsevier Science.

Wysotzki, F., & Schmid, U. (to appear). *Macro-induction for the construction of recursive program schemes from finite example programs* (Tech. Rep.). Dept. of Computer Science, TU Berlin, Germany.

Appendix A

Implementation Details

A.1 Modules of DPlan

- `dplan.lsp`: main program
  construction of minimal spanning tree (`dplan` calls `(ms-tree <current-state>)`) or dag (`dplan` calls `(ms-tree <current-state>)`), plan is saved in `plan-steps` as list of `pstep`-structures;

- `plan-dstruc.lsd`: global data structure `pstep`, see below

- `ps-back.lsp`: calculating the pre-image of the current state (match and backward operator application)
  function `(apply-rules <current-state>)` is called from `dplan`, returns list of new `pstep`'s

- `showplan.lsp`: graphical and term output of plans
  function `(show plan-steps)` is called from `dplan`; graphs can be displayed with `graphlet`, trees can be displayed with `xterm`

A.2 `Pstep`-Data Structure

```
(defstruct pstep
  instop  ; instantiated operator, cf. puttable(A)
  parent  ; parent node: in backward planning successor of instop
          ; input in ps-back "state"
  child   ; child node: in backward planning predecessor of instop
          ; constructed in ps-back
  prec    ; instantiated preconditions of operator
          ; for conditioned operators: union of global and specific
```

```
          ; precondition
  add     ; instantiated add-list
  del     ; instantiated del-liste
  nodeid  ; identifier
  level   ; level in the ms-dag
)
```

A `pstep` is generated in `ps-back.lsp`, `nodeid` and `level` are instantiated in `dplan`.

A.3 Global Structures for Plan Transformation

```
; input from dplan.lsp is a plan saved in plan-steps as a list of psteps

; global variables
(setq subplanlist nil) ; list of all transformed subplans
                       ; (special case: one subplan)
(setq planstruc nil)   ; structure of global plan (subtrees replaced by
                       ; subplan names)

(setq mstlist nil)     ; list of possible minimal spanning trees
                       ; (backtrack-point for dags which are not sets!)
; ..............................................................
; new structure for transformed plan (one for each subplan)
; initial generation in decompose

(defstruct tplan
  pname    ; name of plan (decompose)
  suplan   ; plan as structure (plan-steps) (decompose)
  coplan   ; plan with data types and relevant predicates
  term     ; plan as term
  ptype    ; type of the plan (singleop, seq, set, list)
  newdat   ; *newly constructed datastructure (a list/set of objects)
  newpred  ; *newly constructed predicate (if newdat =/= nil) as pattern
  npfct    ; *function definition for new predicate
  goalpred ; goal-predicate (might be newpred)
  bottom   ; bottom-element (might be newdat)
  passocs  ; const/constr rewrite pairs
  pafct    ; function definition for the rewriting
  pparams  ; input parameters
  pvalues  ; initial values
)
;; newdat, newpred, npfct are not filled for ptype = singleop and ptype = set
;; newpred: p* (... CO ...)  with CO as place-holder for complex object
;;                           maybe additional constant arguments
;;     f.e.: (at* CO B) for rocket
;; bottom == newdat (-> newdat might be superfluous)
```

A.4 Main Components of `plan-transform.lsp`

Plan transformation is implemented in `plan-transform.lsp`. The main function is `(plantransform <plan>)`:

- Input: plan as list of `psteps` from `dplan.lsp`
- `(decompose plan)`:
  initial decomposition; generation and initialization of the global variables `subplanlist` (list of `tplan` structures), `planstruc` (sub-plan structure as term of sub-plan names)
- `(intro-type subplanlist)`:
  data type inferece for each sub-plan, successively filling the slots of `tplan`
- `ptransform subplanlist)`:
  introducing situation variables and rewriting into conditional expression
- Output: transformation information as given in `subplanlist` and `planstruc`; `tplan.term` is passed to the generalization-to-n algorithm for each sub-plan; `planstruc` is extended from sub-plan names to arguments and rewritten into a "main" program

A.4.1 Plan Decomposition

Please note: Extracts from the program are given in an abbreviated pseudo-Lisp notation, omitting implementation details!

```
; call decompose with complete plan and level = 0 (root)
(decompose plan level) ==
(mapcar $\lambda$x.(r-dec-p (get-first-op plan lv) x lv) (partition plan))

(get-first-op plan level) ==
get the set first operator-symbol at the upper-most level of the plan

(partition plan) == for each root of ``plan'', return its subplan

(r-dec-p op plan lv) ==
    (let ((dlv (disag-level op plan))
          (pname (gensym "P")))
       (cond ((not dlv) (save-subplan pname plan)) ; single plan
                                                   ; subplan
             ((> dlv lv) (save-subplan pname (get-suplan dlv plan))
```

```
                         (decompose (append (make-root (1- dlv) plan)
                                            (rest-suplan dlv plan))
                                    dlv)
             )
             (T ; (= dlv lv)                         ; different operators
                         (save-subplan pname plan)   ; at one level
             )           ; currently not treated by decomposition
))

(disag-level op plan) ==
returns the level in ``plan'' on which appears for the first time a
different operator-name than ``op''

(save-suplan pname plan) ==
generate a new tplan-structure and instantiate the slots pname with
``pname'' and suplan with ``plan'' (see appendix for tplan-structure)

(get-suplan dlv plan) ==
return a partial plan from the current top-level to level ``dlv''

(make-root lv plan) == get the leaf of the newly generated subplan

(rest-suplan dlv plan) ==
return the remaining plan with level ``dlv'' as new top-level
```

A.4.2 Data Type Inference

The algorithms for data type inference are discussed in detail in chapter 4.

A.4.3 Introduction of Situation Variables

For each subplan:

```
(rewrite-to-term coplan) ==
   (list 'if (intro-s (first-state coplan))
         's  (r-rewrite (rest coplan))      )

(intro-s e) == extend e by an argument ``s''

(r-rewrite coplan) ==
   (cond ((null cp) 'omega)
         (T (append (intro-s (first-op coplan))
                 (list (list 'if
                   (intro-s (first-state coplan))
                   's (r-rewrite (rest coplan))  )))))
```

A.5 Number of MSTs in a DAG

To calculate the number of minimal spanning trees in an DAG, we can use formula $\prod_{i=0}^{n} c_i$, multiplying the number of alternative choices c_i on each level i. For the sorting of three elements, we have: $1 \cdot 1 \cdot 9 = 9$, with a single option for the root and the first level and 9 different options for the third level. To calculate the number of options on a level, the connective structure of the DAG has to be know. The different alternatives on one level can be calculated as described for the construction of minimal spanning trees. For sorting lists with 4 elements we have: $1 \cdot 1 \cdot 52488 \cdot 46656 = 2.448.880.128!$

To omit the explicit calculation of options per level, an upper bound estimate is

$$c_i = \binom{a_i}{n_i}$$

with a_i as number of arcs from level $i-1$ to level i and n_i as number of nodes on level i.

We cannot provide a formula for calculating the proportion of regularizable trees to all trees, because regularizability depends on the identity of edge-labels which is variable between domains.

A.6 Extracting Minimal Spanning Trees from a DAG

```
; lst is tree until lv (initially: root) ; sp is all plan-steps on levels > lv
(defun extract-trees (lst sp lv)
  (let* ((tvec (number-of-msts 1 sp (1+ lv)))
         (tcnt (reduce '* tvec)))
     (print `(There are * ,tvec = ,tcnt minimal spanning trees))
     (fresh-line)
     (cond ((> tcnt 576) (write-string "How many trees? <number> ")
                         (setq k (read-number))
                         (fresh-line)
                         (extract-msts (list lst) sp (1+ lv) k))
           (T (write-string "Generate all alternatives")
              (fresh-line)
              (extract-msts (list lst) sp (1+ lv) tcnt))
)))

(defun extract-msts (lst sp lv k)
(print `(Include next level ,lv))
  (let ((lp (remove-if #'(lambda(x) (/= lv (pstep-level x))) sp))
        (rp (remove-if #'(lambda(x) (= lv (pstep-level x))) sp)))
```

```
    (cond ((null lp) nil)
          ((null rp) (first-k k (lift
                     (mapcar #'(lambda(x) (pmerge lst x)) (all-combs lp k)))))
           ; in the last step this is only a "throw-away" of
           ; already calculated trees!
          (T (extract-msts (first-k k
                              (lift (mapcar #'(lambda(x) (pmerge lst x))
                                 (all-combs lp k))))
                           rp (1+ lv) k)
)))       )

(defun pmerge (lst e)
  (cond ((null lst) (list e))  ; should never occur
        ((flatlst lst)  (join lst e))
        (T  (mapcar #'(lambda(x) (join x e)) lst))
))

(defun all-combs (lp k)
  (let* ((dc (make-sset (mapcar #'(lambda(x) (pstep-child x)) lp)))
         (lcs (child-split lp dc k))
         (tlcs (mapcar #'(lambda(x) (generate-trees x (1- (length x)))) lcs)))
    (cart-product (car tlcs) (cdr tlcs) k)
))

(defun child-split (lp dc k)
  (cond ((null dc) nil)
        (T (cons (first-k k (remove-if #'(lambda(x)
                          (not (setequal (pstep-child x) (car dc)))) lp) )
                 (child-split lp (cdr dc) k)))
))

(defun generate-trees (lp cnt)
  (cond ((< cnt 0) nil)
        (T (setq cur-lp (copy-plan lp))
            (cons (nth cnt cur-lp)
                (generate-trees lp (1- cnt))))
))

(defun cart-product (fst rst k)
  (cond ((null rst) fst)
        (T (cart-product (first-k k (comb fst (car rst))) (cdr rst) k))
))

(defun comb (f r)
  (cond ((null f) nil)
        (T (append (mapcar #'(lambda(x) (join (car f) x)) r)
                 (comb (cdr f) r)))
))
```

A.7 Regularizing a Tree

```
(defun regularize-tree (mst lv)
  (let ((cur (remove-if #'(lambda(y) (/= lv (pstep-level y))) mst))
        (nxt (remove-if #'(lambda(y) (/= (1+ lv) (pstep-level y))) mst))
       )
    (cond ((null cur) nil)
          ((null nxt) mst)
          (T (let ((opsetcl  (mapcar #'(lambda(x) (pstep-instop x)) cur))
                   (opsetnl (mapcar #'(lambda(x) (pstep-instop x)) nxt)))
               (cond ((subsetp opsetnl opsetcl :test 'equal)
                      (regularize-tree (smerge (lv-shift cur opsetnl)
                                               mst) (1+ lv)))
                     (T (regularize-tree mst (1+ lv)))

)))         )  ) )

(defun lv-shift (cur opsetnl)
  (cond ((null cur) nil)
        ((member (pstep-instop (car cur)) opsetnl :test 'equal)
                  (setf nc (copy-pstep (car cur)))
                  (setf (pstep-parent nc)
                        (id-insert (pstep-parent (car cur))))
                  (setf (pstep-level nc) (1+ (pstep-level (car cur))))
                  (cons
                  (make-pstep
                      :instop 'id
                      :parent (pstep-parent (car cur))
                      :child (id-insert (pstep-parent (car cur)))
                      :nodeid (+ 1000 (pstep-nodeid (car cur)))
                      :level (pstep-level (car cur))
                  )
                  (cons nc (lv-shift (cdr cur) opsetnl)))
         )
       (T (cons (car cur) (lv-shift (cdr cur) opsetnl)))
))

(defun id-insert (s)
  (cond ((null s) nil)
        ((idlist (car s)) (cons (cons 'id (car s)) (cdr s)))
        (T (cons '(id) s))
))

(defun idlist (l)
  (cond ((null l) T)
        ((equal 'id (car l)) (idlist (cdr l)))
        (T nil)
))
```

```
; if pstep-child of new is equal to a pstep-child in old -> keep new
;    (has higher level)
; if pstep-child without "idlist" is equal to a pstep-child in old
;    and both are on the same level -> keep old (the one which has
;    no or a shorter "idlist")
;   ==> parent-node for the nodes in nw with pstep-child of new as
;       parent has to be set to "old" parent!
; these nodes are still in new because of the sequence of
; node construction in lv-shift!
(defun smerge (nw old)
  (cond ((null nw) old)
        ((member (car nw) old :test 'node-equal)
             (smerge (cdr nw) (cons (car nw)
                                (remove-if #'(lambda(x) (node-equal
                                  (car nw) x)) old))))
        ((member (car nw) old :test 'level-equal)
             (smerge (old-parent (car nw) (cdr nw) old) old))
        (T (smerge (cdr nw) (cons (car nw) old)))
))

(defun node-equal (s1 s2)
  (and (= (length (find-if #'(lambda(x) (idlist x)) (pstep-child s1)))
          (length (find-if #'(lambda(x) (idlist x)) (pstep-child s2)))
       )
       (setequal (remove-if #'(lambda(x) (idlist x)) (pstep-child s1))
                 (remove-if #'(lambda(x) (idlist x)) (pstep-child s2))
))     )

(defun level-equal (s1 s2)
  (and (setequal (remove-if #'(lambda(x) (idlist x)) (pstep-child s1))
                 (remove-if #'(lambda(x) (idlist x)) (pstep-child s2))
       )
       (= (pstep-level s1) (pstep-level s2))
))

; if smerge keeps the old state, then the new-state with cld as parent
; has to keept the corresponding old parent
(defun old-parent (cld sl old)
  (cond ((null sl) nil)
        ((setequal (pstep-child cld) (pstep-parent (car sl)))
              (cons (update-par (copy-pstep (car sl))
                                (pstep-child (find-if #'(lambda(x)
                                   (level-equal cld x)) old))) (cdr sl)))
       (T (old-parent cld (cdr sl) old))
))

(defun update-par (sl ud) (setf (pstep-parent sl) ud) sl)
```

Appendix B

Problem-Specific Details

B.1 `TPlan` Structure for *Unstack*

```
#S(TPLAN :PNAME P1
         :SUPLAN
         (#S(PSTEP :INSTOP NIL :PARENT NIL
                   :CHILD
                   ((CLEAR (SUCC (SUCC O3))) (CLEAR (SUCC O3))
                    (CLEAR O3))
                   :PREC NIL :ADD NIL :DEL NIL :NODEID 0 :LEVEL 0)
          #S(PSTEP :INSTOP (UNSTACK (SUCC O3))
                   :PARENT
                   ((CLEAR (SUCC (SUCC O3))) (CLEAR (SUCC O3))
                    (CLEAR O3))
                   :CHILD
                   ((ON O2 O3) (CLEAR (SUCC (SUCC O3)))
                    (CLEAR (SUCC O3)))
                   :PREC ((ON O2 O3) (CLEAR O2)) :ADD ((CLEAR O3))
                   :DEL ((CLEAR O2) (ON O2 O3)) :NODEID 1 :LEVEL 1)
          #S(PSTEP :INSTOP (UNSTACK (SUCC (SUCC O3)))
                   :PARENT
                   ((ON O2 O3) (CLEAR (SUCC (SUCC O3)))
                    (CLEAR (SUCC O3)))
                   :CHILD
                   ((ON O1 O2) (ON O2 O3) (CLEAR (SUCC (SUCC O3))))
                   :PREC ((ON O1 O2) (CLEAR O1)) :ADD ((CLEAR O2))
                   :DEL ((CLEAR O1) (ON O1 O2)) :NODEID 2 :LEVEL 2))
         :COPLAN
         (#S(PSTEP :INSTOP NIL :PARENT NIL :CHILD ((CLEAR O3))
                   :PREC NIL :ADD NIL :DEL NIL :NODEID 0 :LEVEL 0)
          #S(PSTEP :INSTOP (UNSTACK (SUCC O3)) :PARENT ((CLEAR O3))
                   :CHILD ((CLEAR (SUCC O3)))
                   :PREC ((ON O2 O3) (CLEAR O2)) :ADD ((CLEAR O3))
                   :DEL ((CLEAR O2) (ON O2 O3)) :NODEID 1 :LEVEL 1)
          #S(PSTEP :INSTOP (UNSTACK (SUCC (SUCC O3)))
                   :PARENT ((CLEAR (SUCC O3)))
                   :CHILD ((CLEAR (SUCC (SUCC O3))))
                   :PREC ((ON O1 O2) (CLEAR O1)) :ADD ((CLEAR O2))
```

```
          :DEL ((CLEAR O1) (ON O1 O2)) :NODEID 2 :LEVEL 2))
:TERM
(IF (CLEAR O3 S)
    S
  (UNSTACK (SUCC O3) S
   (IF (CLEAR (SUCC O3) S)
       S
     (UNSTACK (SUCC (SUCC O3)) S
      (IF (CLEAR (SUCC (SUCC O3)) S) S OMEGA)))))
:PTYPE SEQ :NEWDAT NIL :NEWPRED NIL :NPFCT NIL
:GOALPRED (CLEAR O3) :BOTTOM O3
:PASSOCS ((O2 (SUCC O3)) (O1 (SUCC (SUCC O3))))
:PAFCT
(DEFUN SUCC (X S)
  (COND ((NULL S) NIL)
        ((AND (EQUAL (FIRST (CAR S)) 'ON)
              (EQUAL (NTH 2 (CAR S)) X))
         (NTH 1 (CAR S)))
        (T (SUCC X (CDR S)))))
:PPARAMS NIL :PVALUES NIL)
```

For *sequences*, the slots `newdat`, `newpred`, and `npfct` are not required. The slots `pparams` and `pvalues` are filled after generalization-to-n.

B.2 The *Rocket* Domain

Please note, that most information for synthesizing control knowledge for *rocket* is given throughout the text, in chapters 2 and 4.

B.2.1 A Lisp-Program for *Rocket*

```
; Control Knowledge for ROCKET
; ----------------------------
; call for example (rocket '(o1 o2 o3) '((at o1 a) (at o2 a) (at o3 a) (at rocket a)))
; or, including generation of oset
; (start-r '((at o1 b) (at o2 b) (at o3 b)) '((at o1 a) (at o2 a) (at o3 a) (at rocket a)))

; predefined set-selectors
(defun pick (oset) (car oset) )
(defun rst (oset) (cdr oset) )

; generalized predicates inferred during plan transformation
(DEFUN AT* (ARGS S)
           (COND ((NULL ARGS) T)
                 ((AND (NULL S) (NOT (NULL ARGS))) NIL)
                 ((AND (EQUAL (CAAR S) 'AT)
                       (INTERSECTION ARGS (CDAR S)))
                  (AT* (SET-DIFFERENCE ARGS (CDAR S)) (CDR S)))
                 (T (AT* ARGS (CDR S)))))
```

```
(DEFUN INSIDE* (ARGS S)
           (COND ((NULL ARGS) T)
                 ((AND (NULL S) (NOT (NULL ARGS))) NIL)
                 ((AND (EQUAL (CAAR S) 'INSIDE)
                       (INTERSECTION ARGS (CDAR S)))
                  (INSIDE* (SET-DIFFERENCE ARGS (CDAR S)) (CDR S)))
                 (T (INSIDE* ARGS (CDR S)))))

; explicit operator application
; in combination with DPlan, the add-del-lists are applied to s
(defun unload (o s)
(print `(unload ,o ,s))
  (cond ((null s) nil)
        ((member o (car s) :test 'equal) (cons (list 'at o 'b) (cdr s)))
        (T (cons (car s) (unload o (cdr s)))) ))
(defun loadr (o s)
(print `(load ,o ,s))
  (cond ((null s) nil)
        ((member o (car s) :test 'equal)
                 (cons (list 'inside o 'rocket) (cdr s)))
        (T (cons (car s) (loadr o (cdr s)))) ))
(defun move-rocket (s)
(print `(move-rocket ,s))
  (cond ((null s) nil)
        ((equal (car s) '(at rocket a)) (cons (list 'at 'rocket 'b) (cdr s)))
        (T (cons (car s) (move-rocket (cdr s)))) ))

; generalized control rules
; abstraction from destination (B) (for at*) and vehicle (Rocket) (for inside*)
(defun unload-all (oset s)
  (if (at* oset s)
      s
      (unload (pick oset) (unload-all (rst oset) s)) ))
(defun load-all (oset s)
  (if (inside* oset s)
      s
      (loadr (pick oset) (load-all (rst oset) s)) ))
(defun rocket (oset s) (unload-all oset (move-rocket (load-all oset s))) )

; "meta"-function, generating the set of objects to be transported
; from the top-level goals
(defun start-r (g s) (rocket (make-co g '(at* CO x)) s))

(defun make-co (goal newpat)
  (cond ((null goal) nil)
        ((string< (string (caar goal)) (string (car newpat)))
             (cons (nth (position 'CO newpat) (car goal))
                   (make-co (cdr goal) newpat)))              ))
```

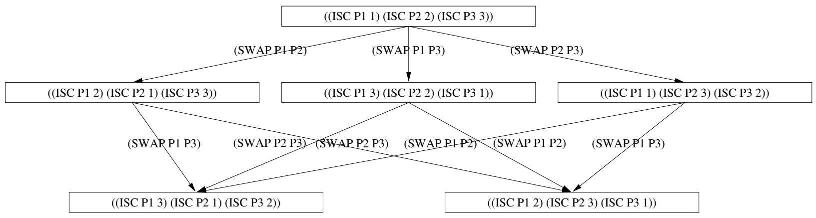


Figure B.1: Universal Plan for Sorting Lists with Three Elements

B.2.2 Interleaving *at* and *inside*

The generalized predicates *(at* oset B)* and *(inside* oset Rocket)* are complements. If all objects are *at* a location, no object is *inside* the vehicle and vice versa. The *unload-all* function presupposes, that all objects in *oset* are inside the rocket and the *load-all* function presupposes, that all objects in *oset* are at the current location. The constrcution of a complex object from the plan is driven by the top-level goals (or for a sub-plan by the predicates in its root-node). After transforming both sub-plans into finite programs and generalizing over them, it becomes clear, that both subplans share the parameter *oset* whith initial value *(o1 o2 o3)*.

Analyzing the relationship between the objects in the *at**-set and the *inside**-set, could lead to an alternative implementation of these generalized predicates: *(at* oset l)* is true, if no object is *inside* the rocket, that means, if for *(inside* oset rocket)* the *oset* is empty (s contains no literal *(inside o rocket)*); and analogous for *(inside* oset rocket)*.

B.3 The *Selection Sort* Domain

B.3.1 Sorting Lists with 3 Elements

Note, that in constructing the universal plan, it is random whether *(swap p q)* or *(swap q p)* is the first instantiation. Because the operator is symmetric, both applications result in an identical state. Only the first application is integrated in the plan. Restriction of swapping only from smaller positions to larger ones (or the other way round) can be done by extending state specifications by `((gt P3 P2) (gt P3 P1) (gt P2 P1))` and the application-condition of the *swap*-operator to `((isc p n1) (isc q n2) (gt n1 n2) (gt q p))`.

B.3.2 Minimal Spanning Trees for *3-SelSort*

From the 9 minimal spanning trees of the 3-sort problem, three are generalizable to the *selsort* program: namely, all trees where the branching factor is as regular as possible (i.e. 3 to 2 vs. 3 to 1).

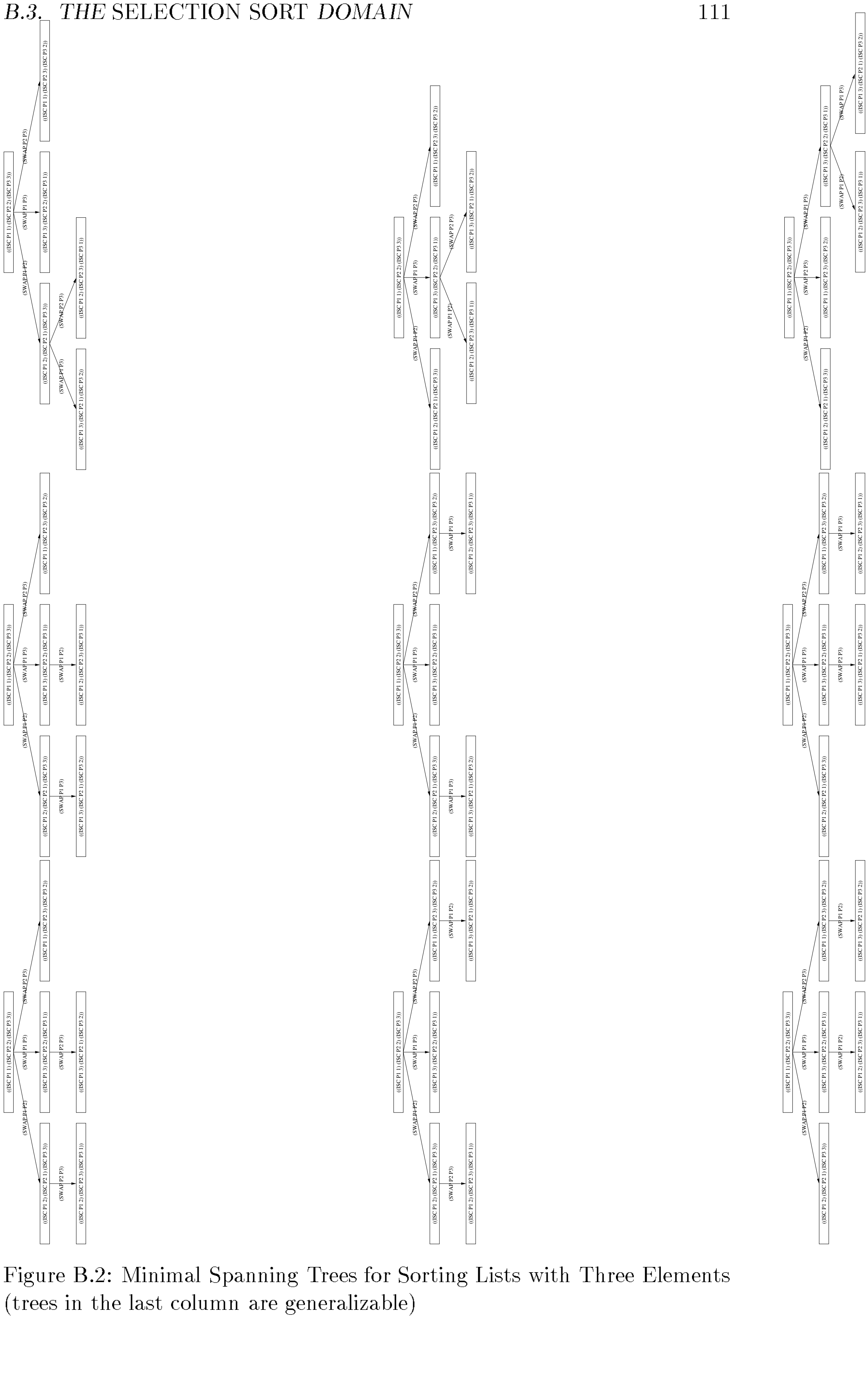


Figure B.2: Minimal Spanning Trees for Sorting Lists with Three Elements (trees in the last column are generalizable)

B.4 The *Tower* Domain

B.4.1 Assuming Subgoal-Independence

```
; abstract representation of control knowledge
;G(n,n,s) = tow(n,n)(s, putt1(n,s))
;G(i,n,s) = tow(i,n)(s,put1(i,s(i),G(s(i),s,n)))
;put1(i,s(i),s) = put(i,s(i),clear(i,clear(s(i),s)))
;putt1(i,s)=putt(i,clear(i,s))
;clear(x,s)=ct(x,s)(s,putt(f(x),clear(f(x),s)))

; main: G(1,n,s)
; -------------------------------------------------------------------
; call (G 1 maxblock s)
; s for maxblock = 3
;   ((on 1 2) (on 2 3) (ct 1) (ont 3))
;   ((on 1 3) (on 3 2) (ct 1) (ont 2))
;   ((on 2 1) (on 1 3) (ct 2) (ont 3))
;   ((on 2 3) (on 3 1) (ct 2) (ont 1))
;   ((on 3 1) (on 1 2) (ct 3) (ont 2))
;   ((on 3 2) (on 2 1) (ct 3) (ont 1))
;   ((on 1 2) (ct 1) (ct 3) (ont 2) (ont 3))
;   ((on 1 3) (ct 1) (ct 2) (ont 2) (ont 3))
;   ((on 2 1) (ct 2) (ct 3) (ont 1) (ont 3))
;   ((on 2 3) (ct 2) (ct 1) (ont 1) (ont 3))
;   ((on 3 1) (ct 3) (ct 2) (ont 1) (ont 2))
;   ((on 3 2) (ct 3) (ct 1) (ont 1) (ont 2))
;   ((ct 1) (ct 2) (ct 3) (ont 1) (ont 2) (ont 3))
; some s for maxblock > 3
;   ((on 4 2) (on 2 1) (on 1 3) (ct 4) (ont 3))
;   ((on 2 5) (on 5 1) (on 4 3) (ct 2) (ct 4) (ont 1) (ont 3))
; -------------------------------------------------------------------
; this would also give true if block n is ont and there are
; unsorted blocks above it!
(defun tow (i n s)
  (cond ((eq i n) (cond ((member (cons 'ont (list n)) s :test 'equal)
                         (print `(tower ,@s)) T)
                         (T nil)
        )         )
        (T        (cond ((and
                (member (cons 'on (list i (1+ i))) s :test 'equal)
                (tow (1+ i) n s)) T)
                         (T nil)
        )         ) ))


(defun putt1 (i s)  (puttable i (clear i s)))
(defun put1 (i s)  (put i (1+ i) (clear i (clear (1+ i) s))) )
```

```
(defun clear (x s)
  (cond ((member (cons 'ct (list x)) s :test 'equal)
                   (print `(block ,@(list x) is clear now)) s)
        (T (print 'puttable-call-by-clear)
           (puttable (f x s) (clear (f x s) s))) ))

(defun put (x y s)
(print s)
 (cond ((null s) (print 'error-in-put) nil)
       ((and (member (cons 'ct (list x)) s :test 'equal)
             (member (cons 'ct (list y)) s :test 'equal)
        ) (print `(put ,@(list x) on ,@(list y) in,@s)) (exec-put x y s))
       (T (print 'error-in-put-xy-not-clear) nil) ))

(defun exec-put (x y s)
  (cond ((null s) nil)
        ((and (equal (first (car s)) 'ont) (equal (second (car s)) x))
                (exec-put x y (cdr s)))
        ((and (equal (first (car s)) 'on) (equal (second (car s)) x))
                (cons (cons 'ct (list (third (car s)))) (exec-put x y (cdr s)))
        )
        ((and (equal (first (car s)) 'ct) (equal (second (car s)) y))
                (cons (cons 'on (list x y)) (exec-put x y (cdr s)))
        )
        (T (cons (car s) (exec-put x y (cdr s)))) ))

(defun puttable (x s)
  (cond ((null s) (print 'error-in-puttable) nil)
        ((member (cons 'ct (list x)) s :test 'equal)
               (print `(puttable ,@(list x) in ,@s)) (exec-puttable x s))
        (T (print 'error-in-puttable-x-not-clear) nil) ))

(defun exec-puttable (x s)
  (cond ((null s) (print 'x-maybe-already-on-table) nil)
        ((and (equal (first (car s)) 'on) (equal (second (car s)) x))
                   (cons (cons 'ct (list (third (car s))))
                           (cons
                           (cons 'ont (list (second (car s))))
                           (cdr s)))
        )
        (T (cons (car s) (exec-puttable x (cdr s)))) ))

(defun f (x s)
  (cond ((null s) nil)
  ((and (equal (first (car s)) 'on) (equal (third (car s)) x))
                                                     (second (car s)))
        (T (f x (cdr s)))   ))

(defun G (i n s)
```

```
  (print `(G ,@(list i) ,@(list n) ,@s))
  (cond ((eq i n) (cond ((tow i n s) s)        ; G(n,n,s)
                             (T (putt1 n s))
        )             )
        (T            (cond ((tow i n s) s)    ; G(i,n,s), i < n
                             (T (put1 i  (G (1+ i) n s)))
        )             )  ))
```

B.4.2 Universal Plan for the 4-Block Tower

The graphics for the universal plan are too large to be included here. They can be viewed online: Use DPlan to generate the graphic output or request the xfig- or eps-file from the author!

In general, the *tower* plan could contain actions *(put x y)* where x is a larger block than y (for example, to reach *(ct d)* from a state *((on b d) (ct a) (ct b) (ct c))* the optimal universal plan can either contain *(puttable b)* or *(put b a)*). A restriction to "ordered" *put*-actions can be reached by presenting *put* before *puttable* in the problem specification – because DPlan applies operators top-down.

B.4.3 Two Programs for *Tower*

```
; TOWER-1; call e.g. (tower '(a b c d) '((on a d) (on d c) (on c b) (ct a)))

(defun tower (olist s)
 (if (subtow olist s)
  s
  (if (and (ct (first olist) s) (subtow (cdr olist) s))
   (put (first olist) (second olist) s)
   (if (and (singleblock olist) (ot (first olist) s))
    (clear-all (first olist) s)
    (if (and (singleblock olist) (ct (first olist) s))
     (puttable (first olist) s)
     (if (singleblock olist)
      (puttable (first olist) (clear-all (first olist) s))
      (if (and (ct (first olist) s) (on* (cdr olist) s))
       (put (first olist) (second olist) (clear-all (second olist) s))
        (if (ct (first olist) s)
          (put (first olist) (second olist) (tower (cdr olist) s))
          (put (first olist) (second olist) (clear-all (first olist)
                                 (tower (cdr olist) s)))
))))))))

; clear-all macro
(defun clear-all (o s )
  (if (ct o s)
```

```
      s
      (puttable (succ o s) (clear-all (succ o s) s ) )) )
(defun succ (x s)
   (cond ((null s) nil)
         ((and (equal (first (car s)) 'on)
               (equal (nth 2 (car s)) x))
          (nth 1 (car s)))
         (T (succ x (cdr s)))))

(defun singleblock (l) (null (cdr l)))

; correct subtower?
(defun subtow (olist s) (and (ct (car olist) s) (on* olist s)))

; tower contains a currect subtower?
(defun on* (olist s)
  (cond ((null olist) T) ; should not happend
        ((and (null (cdr olist)) (ot (car olist) s)) T)
        ((member (list 'on (first olist) (second olist)) s :test 'equal)
        (on* (cdr olist) s))
        (T nil) ))

; given
(defun ct (o s) (member (list 'ct o) s :test 'equal))
; given as (ontable x) OR (here) inferred from state
(defun ot (o s)
  (null (mapcan #'(lambda(x) (and (equal 'on (first x))
                               (equal o (second x))
                               (list x))) s) ))

; explicit application of put-operator
(defun put (x y s )
  (cond ((null s) (print `(put ,x ,y))
                  nil)
        ((equal (car s) (list 'ct y)) (cons (list 'on x y) (put x y (cdr s) )))
        ((and (equal (first (car s)) 'on)
              (equal (second (car s)) x))
                   (cons (list 'ct (third (car s))) (put x y (cdr s) )))
        (T (cons (car s) (put x y (cdr s) )))
))
; explicit application of puttable-operator
(defun puttable (x s )
  (cond ((null s) (print `(puttable ,x))
                  nil)
        ((and (equal (first (car s)) 'on)
             (equal (second (car s)) x)) (cons (list 'ct (third (car s)))
                                          (puttable x (cdr s) )))
        (T (cons (car s) (puttable x (cdr s) )))
))
```

```
; TOWER-2       ;Building a list (tower) of sorted numbers (blocks)
; Ute Schmid Dec 4 97
; ---------------------------------------------------------------
; Representation: 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) #'>)))
))

(defun get-all-no-towers (l max)
 (cond ((null l) nil)
        ((and (equal (bottom (car l)) max) (sorted (get-tower l)))
                                           (get-all-no-towers (cdr l) max))
        ((single-block (get-tower l)) (get-all-no-towers (cdr l) max))
        (T (cons (car l) (get-all-no-towers (cdr l) max)))
))

(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 (maxblock l)))
                         (cdr (get-all-no-towers l (maxblock l)))))
))

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

; 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)))
))
```

List of Tables

List of Figures