# INDUCTIVE SYNTHESIS OF FUNCTIONAL PROGRAMS
## Learning Domain-Specific Control Rules and Abstract Schemes

UTE SCHMID

Fakultät IV – Elektrotechnik und Informatik
Technische Universität Berlin
Deutschland

*Habilitationsschrift – Mai 2001*

**Abstract** In this book a novel approach to inductive synthesis of recursive functions is proposed, combining universal planning, folding of finite programs, and schema abstraction by analogical reasoning. In a first step, an example domain of small complexity is explored by universal planning. For example, for all possible lists over four fixed natural numbers, their optimal transformation sequences into the sorted list are calculated and represented as a DAG. In a second step, the plan is transformed into a finite program term. Plan transformation mainly relies on inferring the data type underlying a given plan. In a third step, the finite program is folded into (a set of) recursive functions. Folding is performed by syntactical pattern-matching and corresponds to inducing a special class of context-free tree grammars. It is shown that the approach can be successfully applied to learn domain-specific control rules. Control rule learning is important to gain the efficiency of domain-specific planners without the need to hand-code the domain-specific knowledge. Furthermore, an extension of planning based on a purely relational domain description language to function application is presented. This extension makes planning applicable to a larger class of problems which are of interest for program synthesis.

As a last step, a hierarchy of program schemes (patterns) is generated by generalizing over already synthesized recursive functions. Generalization can be considered as the last step of problem solving by analogy or programming by analogy. Some psychological experiments were performed to investigate which kind of structural relations between problems can be exploited by human problem solvers. Anti-unification is presented as an approach to mapping and generalizing programs structures. It is proposed that the integration of planning, program synthesis, and analogical reasoning contributes to cognitive science research on skill acquisition by addressing the problem of extracting generalized rules from some initial experience. Such (control) rules represent domain-specific problem solving strategies.

All parts of the approach are implemented in Common Lisp.

**Keywords:** Inductive Program Synthesis, Recursive Functions, Universal Planning, Function Application in Planning, Plan Transformation, Data Type Inference, Folding of Program Terms, Context-free Tree Grammars, Control Rule Learning, Strategy Learning, Analogical Reasoning, Anti-Unification, Scheme Abstraction

*For My Parents and*
*In Memory of My*
*Grandparents*

# Acknowledgments

Research is a kind of work one can never do completely alone. Over the years I profited from the guidance of several professors who supervised or supported my work, namely Fritz Wysotzki, Klaus Eyferth, Bernd Mahr, Jaime Carbonell, Arnold Upmeyer, Peter Pepper, and Gerhard Strube. I learned a lot from discussions with them, with colleagues, and students, such as Jochen Burghardt, Bruce Burns, the group of Hartmut Ehrig, Pierre Flener, Hector Geffner, Peter Geibel, Peter Gerjets, Jürgen Giesl, Wolfgang Grieskamp, Maritta Heisel, Ralf Herbrich, Laurie Hiyakumoto, Petra Hofstedt, Rune Jensen, Emanuel Kitzelmann, Jana Koehler, Steffen Lange, Martin Mühlpfordt, Brigitte Pientka, Heike Pisch, Manuela Veloso, Ulrich Wagner, Bernhard Wolf, Thomas Zeugmann (sorry to everyone I forgot). I am very grateful for the time I could spend at Carnegie Mellon University. Thanks to Klaus Eyferth and Bernd Mahr who motivated me to go, to Gerhard Strube for his support, to Fritz Wysotzki who accepted my absence from teaching, and, of course to Jaime Carbonell who was my very helpful host. My work profited much from the inspiration I got from talks, classes, and discussions, and from the very special atmosphere suggesting that everything is all right as long as "the heart is in the work". I thank all my diploma students who supported the work reported in this book – Dirk Matzke, Rene Mercy, Martin Mühlpfordt, Marina Müller, Mark Müller, Heike Pisch, Knut Polkehn, Uwe Sinha, Imre Szabo, Janin Toussaint, Ulrich Wagner, Joachim Wirth, and Bernhard Wolf. Additional thanks to some of them and Peter Pollmanns for proof-reading parts of the draft of this book. I owe a lot to Fritz Wysotzki for giving me the chance to move from cognitive psychology to artificial intelligence, for many interesting discussions and for critically reading and commenting the draft of this book. Finally, thanks to my colleagues and friends Berry Claus, Robin Hörnig, Barbara Kaup, and Martin Kindsmüller, to my family, and my husband Uwe Konerding for support and high quality leisure time and to all authors of good crime novels.

# Contents

# List of Figures

# List of Tables

Chapter 1

# INTRODUCTION

*She had written what she felt herself called upon to write; and, though she was beginning to feel that she might perhaps do this thing better, she had no doubt that the thing itself was the right thing for her.*

—Harriet Vane in: Dorothy L. Sayers, *Gaudy Night*, 1935

Automatic program synthesis is an active area of research since the early seventies. The application goal of program synthesis is to support human programmers in developing correct and efficient program code and in reasoning about programs. Program synthesis is the core of knowledge based software engineering. The second goal of program synthesis research is to gain more insight in the knowledge and strategies underlying the process of code generation. Developing algorithms for automatic program construction is therefore an area of artificial intelligence (AI) research and human programmers are studied in cognitive science research.

There are two main approaches to program synthesis – deduction and induction. Deductive program synthesis addresses the problem of deriving executable programs from high-level specifications. Typically, the employed transformation or inference rules guarantee that the resulting program is correct with respect to the given specification – but of course, there is no guarantee that the specification is valid with respect to the informal idea a programmer has about what the program should do. The challenge for deductive program synthesis is to provide *formalized knowledge* which allows to synthesize as large a class of programs with as less user-guidance as possible. That is, a synthesis system can be seen as an expert system incorporating general knowledge about algorithms, data structures, optimization techniques, as well as knowledge about the specific programming domain.

Inductive program synthesis investigates program construction from incomplete information, namely from examples for the desired input/output behavior of the program. Program behavior can be specified on different level of detail: as pairs of input and output values, as a selection of computational traces, or as an ordered set of generic traces, abstracting from specific input values. For induction from examples (not to be confused with mathematical proofs by induction) it is not possible to give a notion of correctness. The resulting program has to cover all given examples correctly, but the user has to judge whether the generalized program corresponds to his/her intention.[1] Because correctness of code is crucial for software development, knowledge based software engineering relies on deductive approaches to program synthesis. The challenge for inductive program synthesis is to provide *learning algorithms* which can generalize as large a class of programs with as less background knowledge as possible. That is, inductive synthesis models the ability of extracting structure in the form of – possibly recursive – rules from some initial experience.

This book focusses on *inductive program synthesis*, and more specially on the induction of recursive functions. There are different approaches for learning recursive programs from examples. The oldest approach is to synthesize functional (Lisp) programs. Functional synthesis is realized by a two-step process: In a first step, input/output examples are rewritten into finite terms which are integrated into a finite program. In a second step, the finite program is folded into a recursive function, that is, a program generalizing over finite program is induced. The second step is also called "generalization-to-n" and corresponds to program synthesis from traces or programming by demonstration. It will be shown later that folding of finite terms can be described as a grammar inference problem, namely as induction of a special class of context-free tree grammars. Since the late eighties, there have been two additional approaches to inductive synthesis – inductive logic programming and genetic programming. While the classical functional approach depends on exploiting structural information given in the examples, these approaches mainly depend on search in hypotheses space, that is, the space of syntactically correct programs of a given programming language. The work presented here is in the context of functional program synthesis.

While folding of finite programs into recursive functions can be performed (nearly) by purely syntactical pattern matching, generation of finite terms from input/output examples is knowledge-dependent. The result of rewriting examples – that is, the form and complexity of the finite program – is completely

[1] Please note, that throughout the book I will mostly omit to give both masculine and feminine form and only use masculine for better readability. Also, I will refer to my work in first person plural instead of singular because part of the work was done in collaboration with other people and I do not want to switch between first and third person.

dependent on the set of predefined primitive functions and data structures provided for the rewrite-system. Additionally, the outcome depends on the used rewrite-strategy – 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.

### Overview

In the following, we give a short overview about the different aspects of inductive synthesis of recursive functions which are covered in this book.

**Universal Planning.** We propose to use domain-independent planning to generate finite programs. Inputs correspond to problem states, outputs to states fulfilling the desired goals, and transformation from input to output is realized by calculating optimal action sequences (shortest sequences of function applications). We use universal planning, that is, we consider all possible states of a given finite problem in a single plan. While a "standard" plan represents an (optimal) sequences of actions for transforming one specific initial state into a state fulfilling the given goals, a universal plan represents the set of all optimal plans as a DAG (directed acyclic graph). For example, a plan for sorting lists (or more precisely arrays) of four numbers $1$, $2$, $3$, and $4$, contains all sequences of *swap* operations to transform each of the $4!$ possible input lists into a list *[1 2 3 4]*. Plan construction is realized by a non-linear, state-based, backward algorithm. To make planning applicable to a larger class of problems which are of interest for program synthesis, we present an extension of planning from purely relational domain descriptions to function application.

**Plan Transformation.** The universal plan already represents the structure of the searched-for program, because 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 can be input to a generalization-to-n algorithm. For a program to be generalizable into a (terminating) recursive function, the input states have to be classified with respect to their "size", that is, the objects involved in the planning problem must have an underlying order. We propose a method by which the data type underlying a given plan can be inferred. This information can be exploited in plan transformation to introduce a "constructive" representation of input states, together with an "empty"-test and selector functions.

**Folding of Finite Programs.** Folding of a finite program into a (set of) recursive programs is done by pattern-matching. For inductive generalization, the notion of fix-point semantics can be "inverted": A given finite program is considered as $n$-th unfolding of an unknown recursive program. The term can be folded into a recursion, if it can be decomposed such that each segment of the term matches the same sub-term (called skeleton) and if the instantiations of the skeleton with respect to each segment can be described by a unique substitution (set of replacements of variables in the skeleton by terms). Identifying the skeleton corresponds to learning a regular tree-grammar, identifying the substitution corresponds to an extension of the regular to a context-free tree grammar.

Our approach is independent of a specific programming language: terms are supposed to be elements of a term algebra and the inferred recursive program is represented as a recursive program scheme (RPS) over this term algebra. An RPS is defined as a "main program" (ground term) together with a system of possibly recursive equations over a term algebra consisting of a finite sets of variables, function symbols, and function variables (representing names of user-defined functions). Folding results in the inference of a program *scheme* because the elements of the term algebra (namely the function *symbols*) can be arbitrarily interpreted. Currently, we assume a fixed interpreter function which maps RPSs into Lisp functions with known denotational and operational semantics. We restrict Lisp to its functional core disregarding global variables, variable assignments, loops, and other non-functional elements of the language. As a consequence, an RPS corresponds to a concrete functional program.

For an arbitrarily instantiated finite term which is input to the folder, a term algebra together with the valuation of the identified parameters of the main program is inferred as part of the folding process. The current system can deal with a variety of (syntactic) recursive structures – tail recursion, linear recursion, tree recursion and combinations thereof – and it can deal with recursion over interdependent parameters. Furthermore, a constant initial segment of a term can be identified and used to construct the main program and it is possible to identify sub-patterns distributed over the term which can be folded separately – resulting in an RPS consisting of two or more recursive equations. Mutual recursion and non-primitive recursion are out of the scope of the current system.

**Scheme Abstraction and Analogical Problem Solving.** A set of (synthesized) RPSs can be organized into an abstraction hierarchy of schemes by generalizing over their common structure. For example, an RPS for calculating the factorial of a natural number and an RPS for calculating the sum over a natural number can be generalized to an RPS representing a simple linear recursion over natural numbers. This scheme can be generalized further when regarding simple linear recursive functions over lists. Abstraction is realized

by introducing function variables which can be instantiated by a (restricted) set of primitive function symbols. Abstraction is based on "mapping" two terms such that their common structure is preserved. We present two approaches to mapping – tree transformation and anti-unification. Tree transformation is based on transforming one term (tree) into another by substituting, inserting, and deleting nodes. The performed transformations define the mapping relation. Anti-unification is based on constructing an abstract term containing first and second order variables (i. e., object and function variables) together with a set of substitutions such that instantiating the abstract term results in the original terms.

Scheme abstraction can be integrated into a general model of analogical problem solving (see fig. 1.1): For a given finite program representing some initial experience with a problem, that RPS is retrieved from memory for which its $n$-th unfolding results in a "maximal similarity" to the current (target) problem. Instead of inducing a general solution strategy by generalization-to-n, the retrieved RPS is modified with respect to the mapping obtained between its unfolding and the target. Modification can involve a simple re-instantiation of primitive symbols or more complex adaptations. If an already abstracted scheme is adapted to the current problem, we speak of refinement. To obtain some insight in the pragmatics of analogical problem solving, we empirically investigated what type and degree of structural mapping between two problem structures must exist for successful analogical transfer in human problem solvers.

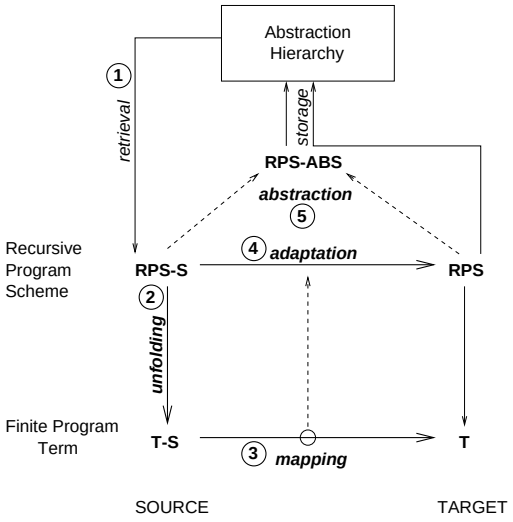



*Figure 1.1.* Analogical Problem Solving and Learning

**Integrating Planning, Program Synthesis, and Analogical Reasoning.** The three components of our approach – planning, folding of finite terms, and analogical problem solving – can be used in isolation or integrated into a complex system. Input in the universal planner is a problem specification represented in an extended version of the standard Strips language, output is a universal plan, represented as DAG. Input in the synthesis system is a finite program term, output is a recursive program scheme. Finite terms can be obtained in arbitrary ways, for example, by hand-coding or by recording program traces. We propose an approach of plan transformation (from a DAG to a finite program term) to combine planning with program synthesis. Generating finite programs by planning is especially suitable for domains, where inputs can be represented as relational objects (sets of literals) and where operations can be described as manipulation of such objects (changing literals). This is true for blocks-world problems and puzzles (as Tower of Hanoi) and for a variety of list-manipulating problems (as sorting). For numerical problems – such as *factorial* or *fibonacci* – we omitt planning and start program synthesis from finite programs generated from hand-coded traces. Input in the analogy module is a finite program term, outputs are the RPS which is most similar to the term, its re-instantiation or adaptation to cover the current term, and an abstracted scheme, generalizing over the current and the re-used RPS. Figure 1.2 gives an overview of the described components and their interactions. All components are implemented in Common Lisp.

### Contributions to Research

In the following, we discuss our work in relation to different areas of research. Our work directly contributs to research in program synthesis and planning. To other research areas, such as software engineering and discovery learning, our work does not directly contribute, but might offer some new perspectives.

**A Novel Approach to Inductive Program Synthesis.** Going back to the roots of early work in functional program synthesis and extending it, exploiting the experience of research available today, results in a novel approach to inductive program synthesis with several advantageous aspects: Adopting the original two-step process of first generating finite programs from examples and then generalizing over them results in a clear separation of the knowledge dependent and the syntactic aspect of induction. Dividing the complex program synthesis problem in two parts allows us to address the sub-problems separately. The knowledge-dependent first part – generating finite programs from examples – can be realized by different approaches, including the rewrite-approach proposed in the seventies. We propose to use state-based planning. While there is a long tradition of combining (deductive) planning and deductive program synthesis, up to now there was no interaction between research on state-based

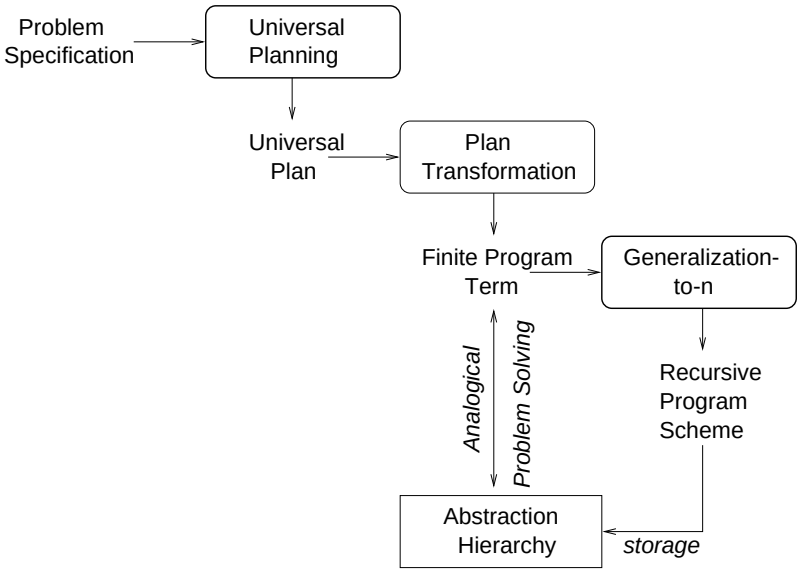



*Figure 1.2.* Main Components of the Synthesis System

planning and research on inductive program synthesis. State-based planning provides a powerful approach to calculate (optimal) transformation sequences from input states to a state fulfilling a set of goal relations by providing a powerful domain specification language together with a domain-independent search algorithm for plan construction. The second part – folding of finite programs – can be solved by a pattern-matching approach. The correspondence between folding program terms and inferring context-free tree grammars makes it possible to give an exact characterization of the class of recursive programs which can be induced. Defining pattern-matching for terms which are elements of an arbitrary term algebra makes the approach independent of a specific programming language. Synthesizing program schemes in contrast to programs allows for a natural combination of induction with analogical reasoning and learning.

**Learning Domain Specific Control Rules for Plans.** Cross-fertilization between state-based planning and inductive program synthesis results in a powerful approach to learning domain specific control rules for plans. Control rule learning currently becomes a major interest in planning research: Although a variety of efficient domain-independent planners have been developed in the nineties, for demanding real world applications it is necessary to guide search for (optimal) plans by exploiting knowledge about the structure of the planning domain. Learning control rules allows to gain the efficiency of

domain-dependent planning without the need to hand-code such knowledge which is a time consuming and error-prone knowledge engineering task. Our functional approach to program synthesis is very well suited for control rule learning because in contrast to logic programs the control flow is represented explicitly. Furthermore, a lot of inductive logic programming systems do not provide an ordering for the induced clauses and literals. That is, evaluation of such clauses by a Prolog interpreter is not guaranteed to terminate with the desired result. In proof planning, as a special domain of planning in the area of mathematical theorem proving and program verification, the need for learning proof methods and control strategies is also recognized. Up to now, we have not applied our approach to this domain, but we see this as an interesting area of further research.

**Knowledge Acquisition for Software Engineering Tools.** Knowledge based software engineering systems are based on formalized knowledge of various general and domain-specific aspects of program development. For that reason, such systems are necessarily incomplete. It depends on the "genius" of the authors of such systems – their analytical insights about program structures and their abilities to explicate these insights – how large the set of domains and the class of programs is that can be supported. Similarly to control rules in planning, tactics are used to guide search. Furthermore, some proof systems provide a variety of proof methods. While execution of transformation or proof steps is performed autonomously, the selection of an appropriate tactic or method is performed interactively. We propose that these components of software engineering tools are candidates for learning. The first reason is, that the guarantee of correctness which is necessary for the fully automated parts of such systems, remains intact. Only the "higher level" strategic aspects of the system which depend on user interaction are subject to learning and the acquired tactics and methods can be accepted or rejected by the user. Our approach to program synthesis might complement these special kind of expert systems by providing an approach to model how some aspects of such expertise develop with experience.

**Programming by Demonstration.** With the growing number of computer users, most of them without programming skills, program synthesis from example becomes relevance for practical applications. For example, watching a users input behavior in a text processing system provides traces which can be generalized to macros (such as "write a letter-head"), watching a users browsing behavior in the world-wide-web can be used to generate preference classes, or watching a users inputs into a graphical editor might provide suggestions for the next actions to be performed. In the simplest case, the resulting programs are just sequences of parameterized operations. By applying inductive program

synthesis, more sophisticated programs, involving loops, could be generated. A further application is to support beginning programmers. A student might interact with an interpreter by first giving examples for the desired program behavior and than watch, how a recursion is formed to generalize the program to the general case.

**Discovery Learning.** Folding of finite programs and some aspects of plan transformation can be characterized as discovery learning. Folding of finite programs models the (human) ability to extract generalized rules by identifying relevant structural aspects in perceptive inputs. This ability can be seen as the core of the flexibility of (human) cognition underlying the acquisition of perceptual categories and linguistic concepts as well as the extraction of general strategies from problem solving experience. Furthermore, we will demonstrate for plan transformation how problem dependent selector functions can be extracted from a universal plan by a purely syntactic analysis of its structure. Because our approach to program synthesis is driven by the underlying structure of some initial experience (represented as universal plan), it is also more cognitively plausible than the search driven synthesis of inductive logic and genetic programming.

**Integrating Learning by Doing and by Analogy.** Cognitive science approaches to skill acquisition from problem solving and early work on learning macro operators from planning focus on combining sequences of primitive operators into more complex ones by merging their preconditions and effects. In contrast, our work addresses the acquisition of problem solving strategies. Because domain dependent strategies are represented as recursive program schemes, they capture the operational aspect of problem solving as well as the structure of a domain. Thereby we can deal with learning by induction and analogy in a unified way – showing how schemes can be acquired from problem solving and how such schemes can be used and generalized in analogical problem solving. Furthermore, the identification of data types from plans captures the evolution of perceptive chunks by identifying the relevant aspects of a problem description and defining an order over such partial descriptions.

### Organization of the Book

The book is organized in three main parts – Planning, Inductive Program Synthesis, and Analogical Problem Solving and Learning.[2] Each part starts with an overview of research, along with an introduction of the basic concepts

[2]Please note, that chapters are numbered arabic and in each chapter sections are numbered arabic, too. To omit confusions when making references to sections, I write "see section $i$ in chapter $j$" if I refer to a different chapter than the current one and otherwise I just write "see section $i$". Figures, tables, definitions,

and formalisms. Afterwards our own work is presented, including relations to other work. We finish with summarizing the contributions of our approach to the field, and giving an outlook to further research. The overview chapters can be read independently of the research specific chapters. The research specific chapters presuppose that the reader is familiar with the concepts introduced in the overview chapters. The focus of our work is on inductive program synthesis and its application to control rule learning for planning. Additionally, we discuss relations to work on problem solving and learning in cognitive psychology.

**Part I: Planning.** In *chapter 2*, first, the standard Strips language for specifying planning domains and problems and a semantics for operator application are introduced. Afterwards, extensions of the Strips language (ADL/PDDL) are discussed and contrasted with situation calculus. Basic algorithms for forward and backward planning are introduced. Complexity of planning and formal properties of planning algorithms are discussed. A short overview of the development of different approaches to planning is given and pre-planning analysis and learning are introduced as methods for obtaining domain specific knowledge for making plan construction more efficient. In *chapter 3*, the non-linear, state-based, universal planner DPlan is introduced and in *chapter 4* an extension of the Strips language to function application is presented. In *chapter 5*, we evaluate our approach and discuss further work to be done.

**Part II: Inductive Program Synthesis.** In *chapter 6* we give a survey of automatic programming research, focussing on automatic program construction, that is, program synthesis. We give a short overview of constructive theorem proving and program transformation as approaches to deductive program synthesis. Afterwards, we present inductive program synthesis as a special case of machine learning. We introduce grammar inference as theoretical background for inductive program synthesis. Genetic programming, inductive logic programming and inductive functional programming are presented as three approaches to generalize recursive programs from incomplete specifications. In *chapter 7* we introduce the concept of recursive program schemes and present our approach to folding finite program terms. In *chapter 8* we present our approach to bridging the gap between planning and program synthesis by transforming plans into program terms. In *chapter 9*, we evaluate our approach and discuss relations to human strategy learning.

---

theorems and so on are numerated chapterwise. That is, a reference to figure $i.j$ refers to the $j$-th figure in chapter $i$.

**Part III: Analogical Problem Solving and Learning.** In *chapter 10* we give an overview of approaches to analogical and case-based reasoning, discussing similarity measures and qualitative concepts for structural similarity. In *chapter 11* some psychological experiments concerning problem solving with non-isomorphic example problems are reported. In *chapter 12* we present anti-unification as an approach to structure mapping and generalization along with preliminary ideas for adaptation of non-isomorphic structures. In *chapter 13* we evaluate our approach and discuss further work to be done.

I

# PLANNING:
# MAKING SOME INITIAL EXPERIENCE WITH A PROBLEM DOMAIN

Chapter 2

# STATE-BASED PLANNING

*Plan it first and then take it.*
—Travis McGee in: John D. MacDonald, *The Long Lavender Look*, 1970

Planning is a major sub-discipline of AI. A plan is defined as a sequence of actions for transforming a given state into a state which fulfills a predefined set of goals. Planning research deals with the formalization, implementation, and evaluation of algorithms for constructing plans. In the following, we first (sect. 2.1) introduce a language for representing problems called standard Strips. Along with defining the language, the basic concepts and notations of planning are introduced. Afterwards (sect. 2) some extensions to Strips are introduced and situation calculus is discussed as an alternative language formalism. In section 3 basic algorithms for plan construction based on forward and backward search are introduced. We discuss complexity results for planning and give results for termination, soundness, and completeness of planning algorithms. In section 4 we give a short survey of well-known planning systems and an overview of concepts often used in planning literature and pointers to literature. Finally (sect. 5), pre-planning analysis and learning are introduced as approaches for acquisition of domain-specific knowledge which can be used to make plan contruction more efficient. The focus is on state-based algorithms, including universal planning. Throughout the chapter we give illustrations using blocks-world examples.

## 1 STANDARD STRIPS

To introduce the basic concepts and notations of (state-based) planning, we first review Strips. The original Strips was proposed by Fikes and Nilsson (1971) and up to now it is used with slight modifications or some extensions

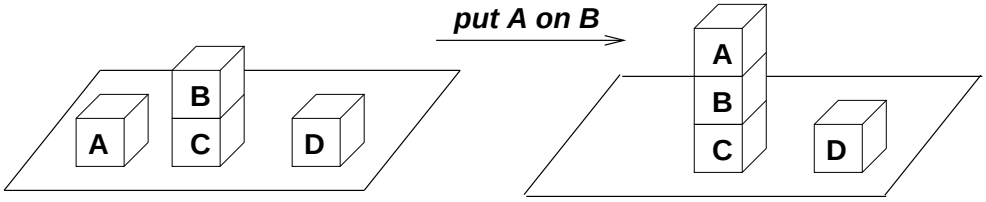



*Figure 2.1.* A Simple Blocks-World

by the majority of planning systems. The main advantage of Strips is that it has a strong expressive power (mainly due to the so called closed world assumption, see below) and at the same time allows for efficient planning algorithms. Informal introductions to Strips are given in all AI text books, for example in Nilsson (1980) and Russell and Norvig (1995).

## 1.1 A BLOCKS-WORLD EXAMPLE

Let us look at a very restricted world, consisting of four distinct blocks. The blocks, called *A*, *B*, *C*, and *D*, are the objects of this blocks-world domain. Each state of the world can be described by the following relations: a block can be on the table, that is, the proposition *ontable(A)* is true in a state, where block *A* is lying on the table; a block can be clear, that is, the proposition *clear(A)* is true in a state, where no other block is lying on top of block *A*; and a block can be lying on another block, that is, the proposition *on(B, C)* is true in a state, where block *B* is lying immediately on top of block *C*. State changes can be achieved by applying one of the following two operators: putting a block on the table and putting a block on another block. Both operators have application conditions: A block can only be moved if it is clear and a block can only be put on another block, if this block is clear, too. Figure 2.1 illustrates this simple blocks-world example. If the goal for which a plan is searched is that *A* is lying on *B* and *B* is lying on *C*, the right-hand state is a goal state, because both proposition *on(A, B)* and proposition *on(B, C)* are true. Note that other states, for example a tower with *D* on top of *A*, *B*, *C*, also fulfill the goal because it was not demanded that *A* has to be clear or that *C* has to be on the table.

To formalize plan construction, a language (or different languages) for describing states, operators, and goals must be defined. Furthermore, it has to be defined how a state change can be (syntactically) calculated.

## 1.2 BASIC DEFINITIONS

The Strips language is defined over literals with constant symbols and variables as arguments. That is, no general terms (including function symbols)

are considered. We use the notion of term in the following definition in this specific way. In chapter 8 we will introduce a language allowing for general terms.

**Definition 2.1 (Strips Language)** *The Strips language $\mathcal{L}_{\mathcal{S}}(\mathcal{X}, \mathcal{C}, \mathcal{R})$ is defined over sets of variables $\mathcal{X}$, constant symbols $\mathcal{C}$, and relational symbols $\mathcal{R}$ in the following way:*

- *Variables $x \in \mathcal{X}$ are terms.*
- *Constant symbols $c \in \mathcal{C}$ are terms.*
- *If $p \in \mathcal{R}$ is a relational symbol with arity $\alpha(p) = i$ and if $t_1, \ldots, t_i$ are terms, then $p(t_1, \ldots, t_i)$ is a formula.*
  *Formulas consisting of a single relational symbol are called (positive) literals. For short, we write $p(\vec{t})$.*
- *If $p_1(\vec{t_1}), \ldots, p_n(\vec{t_n})$ are formulas, then $\{p_1(\vec{t_1}), \ldots, p_n(\vec{t_n})\}$ is a formula, representing the conjunction of literals.*
- *There are no other Strips formulas.*

*Formulas over $\mathcal{L}_{\mathcal{S}}(\mathcal{C}, \mathcal{R})$, i. e., formulas not containing variables are called ground formulas. A literal without variables is called atom.*
*We write $\mathcal{L}_{\mathcal{S}}$ as abbreviation for $\mathcal{L}_{\mathcal{S}}(\mathcal{X}, \mathcal{C}, \mathcal{R})$. With $\mathcal{X}(F)$ we denote the variables occurring in formula $F$.*

For the blocks-world domain given in figure 2.1, $\mathcal{L}_{\mathcal{S}}$ is defined over the following set of symbols: $\mathcal{C} = \{A, B, C, D\}$, $\mathcal{R} = \{on^2, clear^1, ontable^1\}$, where $p^i$ denotes a relational symbol of arity $i$. We will see below that for defining operators additionally a set of variables $\mathcal{X} = \{?x, ?y, ?z\}$ is needed.

The Strips language can be used to *represent* states[1] and to define *syntactic rules* for transforming a state representation by applying Strips operators. A state representation can be *interpreted logically* by providing a domain, i. e., a set of objects of the world, and a denotation for all constant and relational symbols. A state representation denotes all states of the world where the interpreted Strips relations are true. That is, a state representation denotes a family of states in the world. When interpreting a formula, it is assumed that all relations not given explicitly are false (the *closed world assumption*).

[1] Note that we do not introduce different representations to discern between syntactical expressions and their semantic interpretation. We will speak of constant or relation *symbols* when referring to the syntax and of constants or relations when referring to the semantics of a planning domain.

**Definition 2.2 (State Representation)** *A problem state $s$ is a conjunction of atoms. That is, $s \in \mathcal{L}_{\mathcal{S}}(\mathcal{C}, \mathcal{R})$.*

That is, states are propositions (relations over constants).

Examples of problem states are

$s_1$ = {*on(B, C), clear(A), clear(B), clear(D), ontable(A), ontable(C), ontable(D)*}
$s_2$ = {*on(A, B), on(B, C), clear(A), clear(D), ontable(C), ontable(D)*}.

An interpretation of $s_1$ results in a state as depicted on the left-hand side of figure 2.1, an interpretation of $s_2$ results in a state as depicted on the right-hand side.

The relational symbols *on(x, y)*, *clear(x)*, and *ontable(x)* can change their truth values over different states. There might be further relational symbols, denoting relations which are constant over all states, as for example the color of blocks (if there is no operator *paint-block*). Relational symbols which can change their truth values are called fluents, constant relational symbols are called statics. We will see below, that fluents can appear in operator preconditions and effects, while statics can only appear in preconditions.

Before introducing goals and operators defined over $\mathcal{L}_{\mathcal{S}}(\mathcal{X}, \mathcal{C}, \mathcal{R})$, we introduce matching of formulas containing variables (also called patterns) with ground formulas.

**Definition 2.3 (Substitution and Matching)** *A substitution is a set of mappings $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ defining replacements of variables $x_i$ by terms $t_i$. For language $\mathcal{L}_{\mathcal{S}}$ terms $t_i$ are restricted to variables and constants. By applying a substitution $\sigma$ to a formula $F$ – denoted $F_\sigma$ – all variables $x_i \in \mathcal{X}(F)$ with $x_i \leftarrow t_i \in \sigma$ are replaced by the associated $t_i$. Note that this replacement is unique, i. e., identical variables are replaced by identical terms. We call $F_\sigma$ an instantiated formula if all $\mathcal{X}(F)$ are replaced by constant symbols.*
*For a formula $F \in \mathcal{L}_{\mathcal{S}}$ and a set of atoms $A \in \mathcal{L}_{\mathcal{S}}(\mathcal{C}, \mathcal{R})$,* match(F,A) = $\Sigma$ *gives all substitutions $sigma_i \in \Sigma$ with $F_{\sigma_i} \subseteq A$.*

For state $s_1$ given above, the formula $F$ = {*ontable(x), clear(x), clear(y)*} can be instantiated to $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ with $\sigma_1 = \{x \leftarrow A, y \leftarrow B\}$, $\sigma_2 = \{x \leftarrow A, y \leftarrow D\}$, $\sigma_3 = \{x \leftarrow D, y \leftarrow A\}$, $\sigma_4 = \{x \leftarrow D, y \leftarrow B\}$.

For the quantor-free language $\mathcal{L}_{\mathcal{S}}(\mathcal{X}, \mathcal{C}, \mathcal{R})$, all variables occurring in formulas are assumed to be bound by existential quantifiers. That is, all formulas correspond to conjunctions of propositions.

**Definition 2.4 (Goal Representation)** *A goal $\mathcal{G}$ is a conjunction of literals. That is, $\mathcal{G} \in \mathcal{L}_{\mathcal{S}}(\mathcal{C}, \mathcal{R})$.*

An example of a planning goal is $\mathcal{G}$ = {*on(A, B), on(B, C)*}. All states $s$ with $\mathcal{G}_\sigma \subseteq s$ are goal states.

**Definition 2.5 (Strips Operator)** *A Strips operator* op *is described by preconditions PRE, ADD- and DEL-lists*[2]*, with PRE, ADD, DEL* $\in \mathcal{L}_{\mathcal{S}}$*. ADD and DEL describe the operator effect. An instantiated operator* $o = op_{\sigma} \in \mathcal{L}_{\mathcal{S}}(\mathcal{C}, \mathcal{R})$ *is called* action*. We write* $PRE(o)$*,* $ADD(o)$*,* $DEL(o)$ *to refer to the precondition, ADD-, or DEL-list of an (instantiated) operator.*

Operators with variables are also called operator schemes.

An example for a Strips operator is:

| | |
|---|---|
| **Operator:** | put(?x, ?y) |
| **PRE:** | {ontable(?x), clear(?x), clear(?y)} |
| **ADD:** | {on(?x, ?y)} |
| **DEL:** | {ontable(?x), clear(?y)} |

This operator can, for example, be instantiated to

| | |
|---|---|
| **Operator:** | put(A, B) |
| **PRE:** | {ontable(A), clear(A), clear(B)} |
| **ADD:** | {on(A, B)} |
| **DEL:** | {ontable(A), clear(B)} |

We will see below (fig. 2.2), that a second variant for the *put* operator is needed for the case that block $x$ is lying on another block $z$. In a blocks-world with additional relations *green(A), green(B), red(C), red(D),* we could restrict the application conditions for *put* further, for example such, that only green blocks are allowed to be moved. Assuming that a *put* action does not affect the color of a block, *red(x)* and *green(x)* are static symbols.

A usual restriction of operator instantiation is, that different variables have to be instantiated with different constant symbols.[3]

**Definition 2.6 ((Forward) Operator Application)** *For a state* $s$ *and an instantiated operator* $o$*, operator application is defined as* $Res(o, s) = s \setminus DEL(o) \cup ADD(o)\ if\ PRE(o) \subseteq s$*.*

Note that subtracting $DEL(o)$ and adding $ADD(o)$ are commutative (resulting in the same successor state), only if $DEL(o) \cap ADD(o) = \emptyset$, $DEL(o) \subseteq s$, and $ADD(o) \cap s = \emptyset$. The ADD-list of an operator might contain free variables, i. e., variables which do not occur as arguments of the relational symbols in the precondition. This means that matching might only result in partial instantiations. The remaining variables have to be instantiated from the set of constant symbols $\mathcal{C}$ given for the current planning problem.

Operator application gives us a syntactic rule for changing one state representation into another one by adding and subtracting atoms. On the semantic

[2] More exactly, the literals given in ADD and DEL are sets.

[3] The planning language PDDL (see sect. 2) allows explicit use of equality and inequality constraints, such that different variables can be instantiated with the same constant symbol if no inequality constraint is given.

| **Operators:** | |
|---|---|
| | **put(?x, ?y)** |
| **PRE:** | {ontable(?x), clear(?x), clear(?y)} |
| **ADD:** | {on(?x, ?y)} |
| **DEL:** | {ontable(?x), clear(?y)} |
| | **put(?x, ?y)** |
| **PRE:** | {on(?x, ?z), clear(?x), clear(?y)} |
| **ADD:** | {on(?x, ?y), clear(?z)} |
| **DEL:** | {on(?x, ?z), clear(?y)} |
| | **puttable(?x)** |
| **PRE:** | {clear(?x), on(?x, ?y)} |
| **ADD:** | {ontable(?x), clear(?y)} |
| **DEL:** | {on(?x, ?y)} |
| | |
| **Goal:** | {on(A, B), on(B, C)} |
| **Initial State:** | {on(D, C), on(C, A), clear(D), clear(B), ontable(A), ontable(B)} |

*Figure 2.2.* A Strips Planning Problem in the Blocks-World

side, operator application describes state transitions in a state space (Newell and Simon, 1972; Nilsson, 1980), where a relation $s' = Res(o, s)$ denotes that a state $s$ can be transformed into a state $s'$. (Syntactic) operator application is admissible, if $s' = Res(o, s)$ holds in the state space underlying the given planning problem, i. e., if (1) $s'$ denotes a state in the world and if (2) this state can be reached by applying the action characterized by *o* in state $s$.

For the left-hand state in figure 2.1 ($s_1$) and the instantiated *put* operator given above, $PRE(o) \subseteq s$ holds and *Res(o,s)* results in the right-hand state in figure 2.1 ($s_2$).

A Strips *domain* is given as set of operators. Extensions of Strips such as PDDL allow inclusion of additional information, such as types (see sect. 2). A Strips *planning problem* is given as $P(\mathcal{O}, \mathcal{I}, \mathcal{G})$ with $\mathcal{O}$ as set of operators, $\mathcal{I}$ as set of initial states and $\mathcal{G}$ as set of top-level goals. An example for a Strips planning problem is given in figure 2.2.

While the language, in which domains and problems can be represented is clearly defined, there is no unique way of modeling domains and problems. In figure 2.2, for example, we decided, to describe states with the relations *on*, *ontable*, and *clear*. There are two different *put* operators, one is applied if block $x$ is lying on the table, and the other if block $x$ is lying on another block $z$.

An alternative representation of the blocks-world domain is given in figure 2.3. Here, not only the blocks, but also the table are considered as objects of the domain. Unary static relations are used to represent that constant symbols *A*, *B*, *C*, *D* are of type "block". Now all operators can be represented as *put(x, y)*. The first variant describes what happens if a block is moved from the table

| | |
|---|---|
| **Operators:** | |
| | **put(?x, ?y)** |
| **PRE:** | {on(?x, Table), block(?x), block(?y), clear(?x), clear(?y)} |
| **ADD:** | {on(?x, ?y)} |
| **DEL:** | {on(?x, Table), clear(?y)} |
| | **put(?x, ?y)** |
| **PRE:** | {on(?x, ?z), block(?x), block(?y), block(?z), clear(?x), clear(?y)} |
| **ADD:** | {on(?x, ?y), clear(?z)} |
| **DEL:** | {on(?x, ?z), clear(?y)} |
| | **put(?x, Table)** |
| **PRE:** | {clear(?x), on(?x, ?y), block(?x), block(?y)} |
| **ADD:** | {on(?x, Table), clear(?y)} |
| **DEL:** | {on(?x, ?y)} |
| | |
| **Goal:** | {on(A, B), on(B, C)} |
| **Initial State:** | {on(D, C), on(C, A), on(A, Table), on(B, Table), clear(D), clear(B), block(A), block(B), block(C), block(D)} |

*Figure 2.3.* An Alternative Representation of the Blocks-World

and put on another block, the second variant describes, how a block is moved from one block on another, and the third variant describes how a block is put on the table. Another alternative would be, to represent *put* as ternary operator *put(block, from, to)*.

Another decision which influences the representation of domains and problems is with respect to the level of detail. We have completely abstracted from the agent who executes the actions. If a plan is intended for execution by a robot, it becomes necessary to represent additional operators, as picking up an object and holding an object and states have to be described with additional literals, for example what block the agent is currently holding (see fig. 2.4).

## 1.3 BACKWARD OPERATOR APPLICATION

The task of a planning algorithm is, to calculate sequences of transformations from states $s_0 \in \mathcal{I}$ to states $s_G$ with $\mathcal{G} \subseteq s_G$. The planning problem is solved if such a transformation sequence – called a plan – is found. Often, it is also required that the transformations are optimal. Optimality can be defined as minimal number of actions or – for operators associated with different costs – as an action sequence with a minimal sum of costs.

Common to all state-based planning algorithms is that plan construction can be characterized as search in the state space. Each planning step involves the selection of an action. Finding a plan in general involves backtracking over such selections. To guarantee termination for the planning algorithm, it is necessary to keep track of the states already constructed, to avoid cycles.

Search in the state space can be performed *forward* – from an initial state to a goal state –, or *backward* – from a goal state to an initial state. Backward planning is based on backward operator application:

**Definition 2.7 (Backward Operator Application)** *For a state $s$ and an instantiated operator $o$, backward operator application is defined as* $Res^{-1}(o, s) = s \setminus ADD(o) \cup (DEL(o) \cup PRE(o)) \; if \; ADD(o) \subseteq s$.

For state

$s_2$ = {*on(A, B), on(B, C), clear(A), clear(D), ontable(C), ontable(D)*}

backward application of the instantiated operator *put(A, B)* given above results in

$s_2 \setminus$ {*on(A, B)*} $\cup$ {*ontable(A), clear(A), clear(B)*} =
{*on(B, C), clear(A), clear(B), clear(D), ontable(A), ontable(C), ontable(D)*} = $s_1$.

Backward operator application is sound, if for $Res^{-1}(o, s) = s'$ holds $Res(o, s') = s$ for all states of the domain. Backward operator application is complete if for all $Res(o, s') = s$ holds $Res^{-1}(o, s) = s'$ (see sect. 3). Originally, backward operator application was not defined for "complete" state descriptions (i. e., an enumeration of *all* atoms over $\mathcal{R}$ which hold in a current state) but for conjunctions of (sub-) goals. We will discuss so-called *goal regression* in section 3.4.

Note that while plan *construction* can be performed by forward and backward operator applications, plan *execution* is always performed by forward operator application – transforming an initial state into a goal state.

## 2 EXTENSIONS AND ALTERNATIVES TO STRIPS

Since the introduction of the Strips language in the seventies, different extensions have been introduced by different planning groups, both to make planning more efficient and to enlarge the scope to a larger set of domains. The extensions were mainly influenced by work from Pednault (1987, 1994) who proposed ADL (action description language) as a more expressive but still efficient alternative to Strips. The language PDDL (Planning Domain Definition Language) can be seen as a synthesis of all language features which were introduced in the different planning systems available today (McDermott, 1998b).

Strips and PDDL are based on the closed world assumption, allowing that state transformations can be calculated by adding and deleting literals from state descriptions. Alternatively, planning can be seen as logical inference problem. In that sense, a Prolog interpreter is a planning algorithm. Situation calculus as a variant of first order logic was introduced by McCarthy (1963), McCarthy and Hayes (1969). Although most today planning systems are based on the Strips approach, situation calculus is still influential in planning

research. Basic concepts from situation calculus are used to reason about semantic properties of (Strips) planning. Furthermore, deductive planning is based on this representation language.

## 2.1 THE PLANNING DOMAIN DEFINITION LANGUAGE

The language PDDL was developed 1998. Most current planning systems are based on PDDL specifications as input and planning problems used at the AIPS planning competitions are presented in PDDL (McDermott, 1998a; Bacchus et al., 2000). The development of PDDL was a joint project involving most of the active planning research groups of the nineties. Thus, it can be seen as a compromise between the different syntactic representations and language features available in the major planning systems of today.

The core of PDDL is Strips. An example for the blocks-world representation in PDDL is given in figure 2.4. For this example, we included aspects of the behavior of an agent in the domain specification (operators *pickup*, and *putdown*, relation symbols *arm-empty* and *holding*). Note that ADD- and DEL-lists are given together in an effect-slot. All positive literals are added, all negated literals are deleted from the current state.

Extensions of Strips included in PDDL domain specifications are

- Typing,
- Equality constraints,
- Conditional effects,
- Disjunctive preconditions,
- Universal quantification,
- Updating of state variables.

Most modern planners are based on Strips plus the first three extensions. While the first four extensions mainly result in a higher effectiveness for plan construction, the last two extensions enlarge the class of domains for which plans can be constructed.

Unfortunately, PDDL (McDermott, 1998b) does mainly provide a syntactic framework for these features but does give no or only an informal description of their semantics. A semantics for conditional effects and effects with all-quantification is given in (Koehler, Nebel, and Hoffmann, 1997). In the following, we introduce typing, equality constraints, and conditional effects. Updating of state variables is discussed in detail in chapter 4. An example for a blocks-world domain specification using an operator with equality-constraints and conditional effects is given in figure 2.5.

```
(define (domain blocksworld)
(:requirements :strips)
(:predicates (clear ?x)
             (on-table ?x)
             (arm-empty)
             (holding ?x)
             (on ?x ?y))
(:action pickup
  :parameters (?ob)
  :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob))
               (not (arm-empty))))
(:action putdown
  :parameters  (?ob)
  :precondition (holding ?ob)
  :effect (and (clear ?ob) (arm-empty) (on-table ?ob)
               (not (holding ?ob))))
(:action stack
  :parameters  (?ob ?underob)
  :precondition (and (clear ?underob) (holding ?ob))
  :effect (and (arm-empty) (clear ?ob) (on ?ob ?underob)
               (not (clear ?underob)) (not (holding ?ob))))
(:action unstack
  :parameters  (?ob ?underob)
  :precondition (and (on ?ob ?underob) (clear ?ob) (arm-empty))
  :effect (and (holding ?ob) (clear ?underob)
               (not (on ?ob ?underob)) (not (clear ?ob)) (not (arm-empty))))
)

(define (problem tower3)
   (:domain blocksworld)
   (:objects a b c)
   (:init (on-table a) (on-table b)   (on-table c)
          (clear a)  (clear b) (clear c) (arm-empty))
   (:goal (and (on a b) (on b c)))
)
```

*Figure 2.4.* Representation of a Blocks-World Problem in PDDL-Strips

### 2.1.1 TYPING AND EQUALITY CONSTRAINTS

The precondition in figure 2.5 contains equality constraints, expressing, that all three objects involved in the *puton* operator have to be different.

Equality constraints restrict what substitutions are legal in matching a current state and a precondition. That is, definition 2.3, is modified such that for all pairs $(x \leftarrow t), (x' \leftarrow t')$ in a substitution $\sigma$ $t \neq t'$ has to hold, if `(not (= x x'))` is specified in the precondition of the operator.

```
(define (domain blocksworld-adl)
(:requirements :strips :equality :conditional-effects)
(:predicates (on ?x ?y)
             (clear ?x)) ; clear(Table) is static
(:action puton
:parameters (?x ?y ?z)
:precondition (and (on ?x ?z) (clear ?x) (clear ?y)
                               (not  (= ?y ?z)) (not (= ?x ?z))
                                 (not (= ?x ?y)) (not (= ?x Table)))
:effect
              (and (on ?x ?y) (not (on ?x ?z))
                   (when (not (eq ?z Table)) (clear ?z))
                   (when (not (eq ?y Table)) (not (clear ?y)))))
)
```

*Figure 2.5.* Blocks-World Domain with Equality Constraints and Conditioned Effects

Instead of using explicit equality constraints, matching can be defined such that variables with different names must generally be instantiated with different constants. But, explicit equality constraints give more expressive power: giving no equality constraint for two variables $x$ and $y$ allows that these variables can be instantiated by different *or* the same constant. Using "implicit" equality constraints make it necessary to specify two different operators, one with only one variable name ($x = y$), and one with both variable names ($x \neq y$).

Besides equality constraints, types can be used to restrict matching. Let us assume a blocks-world, where movable objects consist of blocks and of pyramids and where no objects can be put on top of a pyramid. We can introduce the following hierarchy of types:

- *table*
- *block*
- *pyramid*
- *movable-object: block, pyramid*
- *flattop-object: table, block.*

Operator *puton(x, y, z)* can now be defined over typed variables *x:movable-object, y:flattop-object, z:flattop-object*. When defining a problem for that extended blocks-world domain, each constant must be declared together with a type.

Typing can be simulated in standard Strips using static relations. A simple example for typing is given in figure 2.3. An example covering the type hierarchy from above is: $\{$*table(T), flattop-object(T), block(B), movable-object(B), flattop-object(B), pyramid(A), movable-object(A)*$\}$. The operator precondi-

tion can then be extended by {*movable-object(x), flattop-object(y), flattop-object(z)*}.

### 2.1.2 CONDITIONAL EFFECTS

Conditional effects allow to represent context dependent effects of actions. In the blocks-world specification given in figure 2.3, three different operators were used to describe the possible effects of putting a block somewhere else (on another block or the table). In contrast, in figure 2.5, only one operator is needed. All variants of *puton(x y z)* have some general application restrictions, specified in the precondition: block $x$ is lying on something ($z$ is another block or the table); and both $x$ and $y$ are clear, where *clear(Table)* is static, i. e., holds in all possible situations. Additionally, the equality constraints in the precondition specify that all objects involved have to be different. Regardless of the context in which *puton* is applied, the result is that $x$ is no longer lying on $z$, but on $y$. This context independent effect is specified in the first line of the effect. The next two lines specify additional consequences: If $x$ was not lying on the table, but on another block $z$, this block $z$ is clear after operator application; and if block $x$ was not put on the table, but on a block $y$, this block $y$ is no longer clear after operator application. Preconditions for conditioned effects are also called secondary preconditions , conditioned effects are also called secondary effects (Penberthy and Weld, 1992; Fink and Yang, 1997).

The main advantage of conditional effects is, that plan construction gets more efficient: In every planning step, *all* operators must be matched with the current state representation and all successfully matched operators are possible candidates for application. If the general precondition of an operator with conditioned effects does not match with the current state, it can be rejected immediately, while for the unconditioned variants given in figure 2.3 all three preconditions have to be checked.

The semantics of applying an operator with conditioned effects is:

**Definition 2.8 (Operator Application with Conditional Effects)** *Let PRE be the general precondition of an operator, and ADD and DEL the unconditioned effects. Let $PRE_i$, $ADD_i$, $DEL_i$, with $i = 0 \ldots n$ be context conditions with their associated context effects. For a state $s$ and an instantiated operator $o$, operator application is defined as* $Res(o,s) = s \setminus (DEL(o) \bigcup_{i \in M} DEL_i(o)) \cup (ADD(o) \bigcup_{i \in M} ADD_i(o))$ $if\ PRE(o) \subseteq s$ *and* $M = \{i \mid PRE_i(o) \subseteq s\}$. *Backward application is defined as* $Res^{-1}(o,s) = s \setminus (ADD(o) \bigcup_{i \in M} ADD_i(o)) \cup [(DEL(o) \bigcup_{i \in M} DEL_i(o)) \cup (PRE(o) \bigcup_{i \in M} PRE_i(o))]$ $if\ ADD(o) \subseteq s$ *and* $M = \{i \mid ADD_i(o) \subseteq s\}$.

## 2.2 SITUATION CALCULUS

Situation calculus was introduced by McCarthy (McCarthy, 1963; McCarthy and Hayes, 1969) to describe state transitions in first order logic. The world is conceived as a sequence of situations and situations are generated from previous situations by actions. Situations are – as Strips state representations – necessarily incomplete representations of the states in the world!

Relations which can change over time are called fluents. Each fluent has an extra argument for representing a situation. For example, *clear(a,* $s_1$*)*[4] denotes, that block $a$ is clear in a situation referred to as $s_1$. Changes in the world are represented by a function *Res(action, situation) = situation*. For example, we can write $s_2$ = *Res(put(a, b),* $s_1$*)* to describe that applying *put(a, b)* in situation $s_1$ results in a situation $s_2$. Note that the concept of fluents is also used in Strips. Although fluents are there not specially marked, it can be inferred from the operator effects, which relational symbols correspond to fluent relations. We already used the *Res*-function to describe the semantics of operator applications in Strips (sect. 2.1). In situation calculus, the result of an operator application is not calculated by adding and deleting literals from a state but by logical inference.

The first implemented system using situation calculus for automated plan construction was proposed by Green (1969) with the QA3 system. Alternatively to the explicit use of a *Res*-function, not only fluents, but also actions are provided with an additional argument for situations. For example, we can write $s_2$ = *put(a, b,* $s_1$*)* to describe that block $a$ is put on block $b$ in situation $s_1$, resulting in a new situation $s_2$. Green's inference system is based on *resolution*.[5] For example, the following two axioms might be given (the first axiom is a specific fact):

**A1** *on(a, table,* $s_1$*)*

**A2** $\forall\, S[on(a, table, S) \rightarrow on(a, b, put(a, b, S))] \equiv$
$\neg$*on(a, table, S)* $\vee$ *on(a, b, put(a, b, S))* *(clausal form).*

Green's theorem prover provides two results: first, it infers, whether some formula – representing a planning goal – follows from the axioms, and second, it provides the action sequence – the plan – if the goal statement can be derived. Not only a yes/no answer but also a plan how the goal can be achieved can be returned because a so called *answer* literal is introduced. The answer literal is initially given as *answer(S)* and at each resolution step, the variable $S$ contained in the answer literal is instantiated in accordance with the involved formulas.

[4] We represent constants with small letters and variables with large letters.

[5] We do not introduce resolution in a formal way but give an illustratory example. We assume that the reader is familiar with the basic concepts of theorem proving, as they are introduced in logic or AI textbooks.

For example, we can ask the theorem prover whether there exists a situation $S_F$ in which *on(a, b, $S_F$)* holds, given the pre-defined axioms. If such a situation $S_F$ exists, *answer($S_F$)* will be instantiated throughout the resolution proof with the plan. Resolution proofs work by contradiction. That is, we start with the negated goal:

1. ¬*on(a, b, $S_F$)* (Negation of the theorem)
2. ¬ *on(a, table, S)* ∨ *on(a, b, put(a, b, S))* (A2)
3. ¬ *on(a, table, S)* (Resolve 1, 2)
   answer(put(a, b, S))
4. *on(a, table, $s_1$)* (A1)
5. contradiction (Resolve 3, 4)
   answer(put(a, b, $s_1$))

The resolution proof shows that a situation $s_2$ = *on(a, table, $s_1$)* with *on(a, b, $s_2$)* exists and that $s_2$ can be reached by putting $a$ on $b$ in situation $s_1$.

Situation calculus has the full expressive power of first order predicate logic. Because logical inference does not rely on the closed world assumption, specifying a planning domain involves much more effort than in Strips: In addition to axioms describing the effects of operator applications, *frame axioms*, describing what predicates remain unaffected by operator applications, have to be specified.

Frame axioms become always necessary when the goal is not only a single literal but a conjunction of literals. For illustration, we extend the example given above:

**A3** *on(a, table, $s_1$)*

**A4** *on(b, table, $s_1$)*

**A5** *on(c, table, $s_1$)*

**A6** ¬*on(X, table, S)* ∨ *on(X, Y, put(X, Y, S))*

**A7** $\forall\, S[on(Y, Z, S) \rightarrow on(Y, Z, put(X, Y, S))] \equiv$
¬*on(Y, Z, S)* ∨ *on(Y, Z, put(X, Y, S))*

**A8** $\forall\, S[on(X, table, S) \rightarrow on(X, table, put(Y, Z, S))] \equiv$
¬*on(X, table, S)* ∨ *on(X, table, put(Y, Z, S))*.

Axiom $A6$ corresponds to axiom $A2$ given above, now stated in a more general form, abstracting from blocks $a$ and $b$. Axiom $A7$ is a frame axiom, stating that a block $Y$ is still lying on a block $Z$, after a block $X$ was put on block

**Effect Axioms:**

| | |
|---|---|
| on(X, Y, put(X, Y, S)) ← | clear(X, S) ∧ clear(Y, S) |
| clear(Z, put(X, Y, S)) ← | on(X, Z, S) ∧ clear(X, S) ∧ clear(Y, S) |
| clear(Y, puttable(X, S)) ← | on(X, Y, S) ∧ clear(X, S) |
| ontable(X, puttable(X, S)) ← | clear(X, S) |

**Frame Axioms:**

| | |
|---|---|
| clear(X, put(X, Y, S)) ← | clear(X, S) ∧ clear(Y, S) |
| clear(Z, put(X, Y, S)) ← | clear(X,S) ∧ clear(Y, S) ∧ clear(Z, S) |
| ontable(Y, put(X, Y, S)) ← | clear(X, S) ∧ clear(Y, S) ∧ ontable(Y, S) |
| ontable(Z, put(X, Y, S)) ← | clear(X, S) ∧ clear(Y, S) ∧ ontable(Z, S) |
| on(Y, Z, put(X, Y, S)) ← | clear(X, S) ∧ clear(Y, S) ∧ on(Y, Z, S) |
| on(W, Z, put(X, Y, S)) ← | clear(X, S) ∧ clear(Y, S) ∧ on(W, Z, S) |
| clear(Z, puttable(X, S)) ← | clear(X, S) ∧ clear(Z, S) |
| ontable(Z, puttable(X, S)) ← | clear(X, S) ∧ ontable(Z, S) |
| on(Y, Z, puttable(X, S)) ← | clear(X, S) ∧ on(Y, Z, S) |
| clear(Z, puttable(X, S)) ← | on(Y, X, S) ∧ clear(Y, S) ∧ clear(Z, S) |
| ontable(Z, puttable(X, S)) ← | on(Y, X, S) ∧ clear(Y, S) ∧ ontable(Z, S) |
| on(W, Z, puttable(X, S)) ← | on(Y, X, S) ∧ clear(Y, S) ∧ on(W, Z, S) |

**Facts (Initial State):**

on(d, c, $s_1$)
on(c, a, $s_1$)
clear(d, $s_1$)
clear(b, $s_1$)
ontable(a, $s_1$)
ontable(b, $s_1$)

**Theorem (Goal):**

on(a, b, S) ∧ on(b, c, S)

*Figure 2.6.* Representation of a Blocks-World Problem in Situation Calculus

$Y$. Axiom $A8$ is a frame axiom, stating that a block $X$ is still lying on the table, if a block $Y$ is put on a block $Z$. Note, that the given axioms are only a sub-set of the information needed for modeling the blocks-world domain. A complete axiomatization is given in figure 2.6 (where we represent the axioms in a Prolog-like notation, writing the conclusion side of an implication on the left-hand side).

For the goal $\exists\, S_F[on(a, b, S_F) \wedge on(b, c, S_F)]$, the resolution proof is[6]:

1. $\neg$*on(a, b, $S_F$)* $\vee$ $\neg$*on(b, c, $S_F$)* (Negation of the theorem)
2. $\neg$*on(X, table, S)* $\vee$ *on(X, Y, put(X, Y, S))* (A6)
3. $\neg$*on(b, c, put(a, b, S))* $\vee$ $\neg$*on(a, table, S)* (Resolve 1, 2)

[6]When introducing a new clause, we rename variables such that there can be no confusion, as usual for resolution.

4. $\neg$*on(Y, Z, S')* $\vee$ *on(Y, Z, put(X, Y, S'))* (A7)
5. $\neg$*on(a, table, S')* $\vee$ $\neg$*on(b, c, S')* (Resolve 3, 4)
6. $\neg$*on(X, table, S)* $\vee$ *on(X, Y, put(X, Y, S))* (A6)
7. $\neg$*on(a, table, put(b, c, S))* $\vee$ $\neg$*on(b, table, S)* (Resolve 5, 6)
8. $\neg$*on(X, table, S')* $\vee$ *on(X, table, put(Y, Z, S'))* (A8)
9. $\neg$*on(b, table, S)* $\vee$ $\neg$*on(a, table, S)* (Resolve 7, 8)
10. *on(b, table,* $s_1$*)* (A4)
11. $\neg$*on(a, table, S)* (Resolve 9, 10)
12. *on(a, table,* $s_1$*)* (A3)
13. contradiction.

Resolution gives us an inference rule for proving that a goal theorem follows from a set of axioms and facts. For automated plan construction, additionally a strategy which guides the search for the proof (i. e., the sequence in which axioms and facts are introduced into the resolution steps) is needed. An example for such a strategy is SLD-resolution as used in Prolog (Sterling and Shapiro, 1986). In general, finding a proof involves backtracking over the resolution steps and over the ordering of goals.

The main reason why Strips and not situation calculus got the standard for domain representations in planning is certainly that it is much more time consuming and also much more error-prone to represent a domain using effect and frame axioms in contrast to only modeling operator preconditions and effects. Additionally, special purpose planning algorithms are naturally more efficient than general theorem provers. Furthermore, for a long time, the restricted expressiveness of Strips was considered sufficient for representing domains which are of interest in plan construction. When more interesting domains, for example domains involving resource constraints (see chap. 4), were considered in planning research, the expressiveness of Strips was extended to PDDL. But still, PDDL is a more restricted language than situation calculus. Due to the progress in automatic theorem proving over the last decade, the efficiency concerns which caused the prominence of state-based planning, might no longer be true (Bibel, 1986). Therefore, it might be of interest again, to compare current state-based and current deductive (situation calculus) approaches

# 3 BASIC PLANNING ALGORITHMS

In general, a planning algorithm is a special purpose search algorithm. State-based planners search in the state-space, as shortly described in section 2.1.

Another variant of planners, called partial-order planners, search in the so-called plan space. Deductive planners search in the "proof space", i. e. the possible orderings of resolution steps. Basic search algorithms are introduced in all introductory algorithm textbooks, for example in Cormen, Leiserson, and Rivest (1990).

In the following, we will first introduce basic concepts for plan construction. Then forward planning is described, followed by a discussion of complexity results for planning and formal properties for plans. Finally, we introduce backward planning.

## 3.1 INFORMAL INTRODUCTION OF BASIC CONCEPTS

The definitions for forward and backward operator application given above (def. 2.6 and def. 2.7) are the crucial component for plan construction: The transformation of a given state into a next state by operator application constitutes one planning step. In general, each planning step consists of matching all operators with the current state description, selecting one instantiated operator which is applicable in the current state, and applying this operator. Plan construction involves a series of such **match-select-apply cycles**. In each planning step *one* operator is selected for application, that is, plan construction is based on depth-first search and operator selection is a backtrack point. During plan construction, a **search tree** is generated. State descriptions are nodes, action applications are arcs in the search tree. Each planning step **expands** the current leaf node $s$ of the search tree by introducing a new action $o$ and the state description $s'$ resulting from applying $o$ to $s$. For forward planning, search starts with an initial state as root; for backward planning, search starts with the top-level goals (or a goal state) as root.

Input in a planning algorithm is a **planning problem** $P(\mathcal{O}, \mathcal{I}, \mathcal{G})$, as defined in section 2.1. Output of a planning algorithm is a **plan**. For basic Strips planning, a plan is a sequence of actions, transforming an initial state into an state fulfilling the top-level goals. Such an executable plan is also called **solution**. More general, a plan is a set of operators together with a set of binding constraints for the variables occurring in the operators and a set of ordering constraints defining the sequence of operator applications. If the plan is not executable, a plan is also called **partial plan**. A plan is not executable if it does not contain all operators necessary to transform an initial state into a goal state, or if not all variables are instantiated, or if there is no complete ordering of the operators.

When implementing a planner, at least the following functionalities must be provided for:

- A pattern-matcher and possibly a mechanism for dealing with free variables (i. e., variables which are not bound by matching an operator with a set of atoms).
- A strategy for selecting an applicable action and for handling backtracking (taking back an earlier commitment).
- A mechanism for calculating the effect of an operator application.
- A data structure for storing the partially constructed plan.
- A data structure for holding the information necessary to detect cycles (i. e., states which were already generated) to guarantee termination.

Some planning systems calculate a set of actions *before* plan construction – by instantiating the operators with the given constant symbols. As a consequence, during plan construction, matching is replaced by a simple subset-test, where it is checked whether the set of instantiated preconditions of an operator is completely contained in the set of atoms representing the current state. Instantiation of operators is realized with respect to the (possibly typed) constants defined for an problem by extracting them from a given initial state or by using the set of explicitly declared constants. In general, such a "freely generated" set of actions is a super-set of the set of "legal" actions.

Often planning algorithms do not directly construct a plan as fully instantiated, totally ordered sequence of actions. Some planners construct partially ordered plans where some actions for which no order constraints were determined during planning are "parallel". Some planners store plans as part of a more general data structure. In such cases, additional functionalities for plan linearization and/or plan extraction must be provided.

## 3.2 FORWARD PLANNING

Plan construction based on forward operator application starts with an initial state as root of the search tree. Plan construction terminates successfully, if a state is found which satisfies all top-level planning goals. Forward planning is also called *progression planning*. An informal forward planning algorithm is given in table 2.1.

In this algorithm, we abstract from the data structure for saving a plan and from the data structure necessary for handling backtracking and for detection of cycles. One possibility would be, to put in each planning step an action-successor-state pair on a stack. When the algorithm terminates successfully, the plan corresponds to the sequence of actions on the stack. For backtracking and cycle detection a second data structure is needed where all generated and rejected states are saved. Both informations can be represented together, if the

*Table 2.1.* Informal Description of Forward Planning

- Until the top-level goals are satisfied in a state or until all possible states are explored DO For the current state $s$:
  - MATCH: For all operators $op \in \mathcal{O}$ calculate all substitutions such that the operator preconditions are contained in $s$. Generate a set of action candidates $A = \{o \mid PRE(o) \subseteq s\}$.
  - SELECT: Select one element $o$ from $A$.
  - APPLY: Calculate the successor state $Res(s, o) = s'$.
- BACKTRACK: If there is no successor state ($A$ is empty), go back to the predecessor state of $s$. If the generated successor state is already contained in the plan, select another element from $A$.
- PROCEED: Otherwise, insert the selected action in the plan and proceed with $s'$ as current state.

complete search history is saved explicitly in a search tree (see Winston, 1992, chap. 4). A search tree can be represented as list of lists:

**Definition 2.9 (The Data Structure "Search Tree")** *A search tree $ST$ can be represented as a list of lists, where each list represents a path: $ST = nil \mid cons(path, ST)$ with*

$$empty(ST) = \begin{cases} true & if\ ST = nil \\ false & else \end{cases}$$

*and selector functions* first(cons(path, ST)) = path, rest(cons(path, ST)) = ST.

- *A (partially expanded) path is defined as a list of action-state pairs $(o\ s)$. The root (initial state) is represented as $(nil\ s)$. The selector function* getaction((o s)) *returns the first, and the selector function* getstate((o s)) *the second argument of an action-state pair.*
- *A path is defined as: $path = nil \mid rcons(path, pair)$ with selector function* last(rcons(path, pair)) = pair.
  *With* getstate(last(path)) *a leaf of the search tree is retrieved. For a current state* s = getstate(last(path)) *and a list of action candidates $A$ a path is expanded by* expand(path, A) = rcons(path, cons(o, s')) *for all $o \in A$ and $Res(o, s) = s'$.*
  *A plan can be extracted from a path with* getplan(path) = map($\lambda$(x). getaction(x) (path)), *where the higher-oder function map describes that function* getaction *is applied to the sequence of all elements in* path.

A "fleshed-out" version of the algorithm in table 2.1 is given in table 2.2. For saving the possible backtrack points now *all* actions which are applicable in a state are inserted in the search tree, together with their corresponding successor states. That is, selection of an action is delayed one step. The selection strategy is to always expand the first ("left-most") path in the search tree. That is, the algorithm is still depth-first. To guarantee termination, for

*Table 2.2.* A Simple Forward Planner

- Main Function *fwplan(O, G, ST)*
- Initial Function Call: *fwplan($\mathcal{O}$, $\mathcal{G}$, [[(nil s)]])* with a set of operators $\mathcal{O}$, an initial state $s \in \mathcal{I}$, and a set of top-level goals $\mathcal{G}$
- Return: A plan as sequence of actions which transform $s$ in a state $s_G$ with $\mathcal{G} \subseteq s_G$, extracted from search tree $ST$.

1. IF *empty(ST)* THEN "no plan found"
2. ELSE LET be current state $S = getstate(last(first(ST)))$.
   (corresponds to SELECT action *getaction(last(first(ST)))*)
   (a) IF $G \subseteq S$ THEN *getplan(first(ST))*
   (b) ELSE
      i. MATCH: For all $op \in O$ calculate $match(op, S)$ as in def. 2.3.
      LET A = $\{o_1, \ldots o_n\}$ be the list of all instantiated operators with $PRE(o_i) \subseteq S$.
      ii. APPLY: For all $o_i \in A$ calculate $S'_i = Res(o_i, S)$ as in def. 2.6.
      LET AR = $\{(o_1\ S'_1) \ldots (o_n\ S'_n)\}$
      iii. Cycle-Test: Remove all pairs $(o_i\ S'_i)$ from $AR$ where $S_i$ is contained in $first(ST)$.
      iv. Recursive call:
      IF *empty(AR)* THEN *fwplan(O, G, rest(ST))* (BACKTRACK)
      ELSE *fwplan(O, G, append(expand(first(ST), AR), tail(ST)))*.

the cycle-test it is sufficient to check whether an action results in a state which is already contained in the current path. Alternatively, it could be checked, if the new state is contained already anywhere else in the search tree. This extended cycle check makes sure that a state is always reached with the shortest possible action sequence but involves possibly more backtracking. Note that the extended cycle-test does not result in *optimal* plans: although every state already included in the search tree is reached with the shortest possible action sequence, the optimal path might be found in a not expanded part of the search tree which does not include this state.

An example for plan construction by forward search is given in figure 2.7. We use the operators as defined in figure 2.2. The top-level goals are *on(A, B), on(B, C)*, and the initial state is *on(A, C), ontable(B), ontable(C), clear(A), clear(B)*. For better readability, the states are presented graphically and not as set of literals. All generated and detected cycles are presented in the figure but only for the first two backtracking is explicitly depicted. The generation of action candidates is based on an ordering of *put* before *puttable* and instantiation is based on alphabetical order (*A* before *B* before *C*).

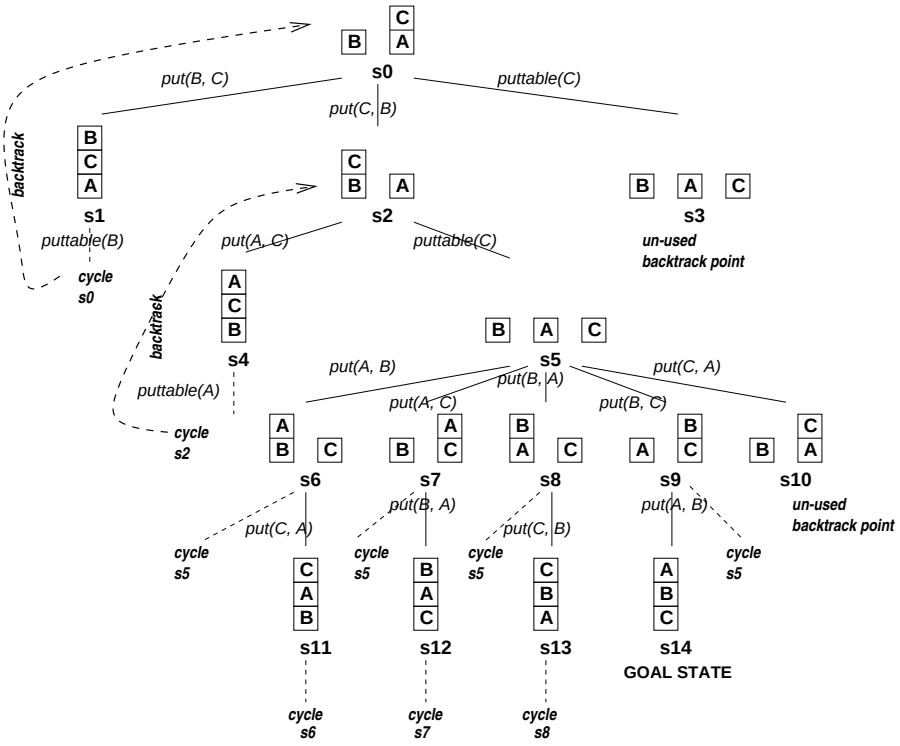



*Figure 2.7.* A Forward Search Tree for Blocks-World

For the given sequence in which action candidates are expanded, search is very inefficient. In fact, all possible states of the blocks-world are generated to find the solution. In general, un-informed forward search has a high branching factor. Forward search can be made more efficient, if some information about the probable distance of a state to a goal state can be used. The best known heuristic forward search algorithm is $A^*$ (Nilsson, 1971), where a lower bound estimate for the distance from the current state to the goal is used to guide search. For domain independent planning – in contrast to specialized problem solving – such a domain specific information is *not* available. An approach how to generate such estimates in a pre-processing step to planning was presented by Bonet and Geffner (1999).

## 3.3 FORMAL PROPERTIES OF PLANNING

### 3.3.1 COMPLEXITY OF PLANNING

Even if a search tree can be kept smaller than in the example given above, in the worst case planning is NP-complete. That is, a (deterministic) algorithm

needs exponential time for finding a solution (or deciding that a problem is not solvable). For a search tree with a maximal depth of $n$ and a maximal branching factor of $m$, in the worst case, planning effort is $m^n$, i. e., the complete search tree has to be generated.[7] Even worse, for problems without a restriction of the maximal length of a possible solution, planning is PSPACE-complete. That is, even for non-deterministic polynomial algorithms an exponential amount of memory is needed (Garey and Johnson, 1979).

Because in the worst case, all possible states have to be generated for plan construction, the number of nodes in the search tree is approximately equivalent to the number of problem states. For problems, where the number of states grows systematically with the number of objects involved, the maximal size of the search tree can be calculated exactly. For example, for the Tower of Hanoi domain[8] with three pegs and $n$ discs, the number of states is $3^n$ (and the minimal depth of a solution path is $2^{n-1}$).

The blocks-world domain is similar to the Tower of Hanoi domain but has less constraints. First, each block can be put on each other block (instead of a disc can only be put on another disc if it is smaller) and second, the table is assumed to have always additional free space for a block (instead of only three pegs where discs can be put). Enumeration of all states of the blocks-world domain with $n$ blocks corresponds to the abstract problem of generating a set of all possible sets of lists which can be constructed from $n$ elements. For example, for three blocks $A$, $B$, $C$:

*{ {(A B C)},{(A C B)}, {(B A C)},{(B C A)},{(C A B)},{(C B A)},*
*{(B C), (A)}, {(C B), (A)}, {(A C), (B)}, {(C A), (B)}, {(A B), (C)}, {(B A), (C)},*
*{(A) (B) (C)} }.*

A single list with three elements represents a single tower, for example a tower with blocks $A$ on $B$ and $B$ on $C$ for the first element given above. A set of two lists represents two towers on the table, for example block $B$ lying on block $C$ and $A$ as a one-block tower. The number of sets of lists corresponds to the so-called Lah-number (Knuth, 1992).[9] It can be calculated by the following formula:

*a(0) = 0*
*a(1) = 1*
*a(n) = [(2n - 1) · a(n - 1)] - [(n - 1) · (n - 2) · a(n - 2)].*

The growth of the number of states for the blocks-world domain is given for up to sixteen blocks in table 2.3.

[7] Note that this only holds for *finite* planning domains where all states are enumerable. For more complex domains involving relational symbols over arbitrary numerical arguments, the search tree becomes infinite and therefore other criteria have to be introduced for termination (making planning incomplete, see below).
[8] The Tower of Hanoi domain is introduced in chapter 4.
[9] More background information can be found at `http://www.research.att.com/cgi-bin/access.cgi/as/njas/sequences/eisA.cgi?Anum=000262`.

*Table 2.3.* Number of States in the Blocks-World Domain

| | | | | | |
|---|---|---|---|---|---|
| # blocks | 1 | 2 | 3 | 4 | 5 |
| # states | 1 | 3 | 13 | 73 | 501 |
| approx. | $1.0 \times 10^0$ | $3.0 \times 10^0$ | $1.3 \times 10^1$ | $7.3 \times 10^1$ | $5.0 \times 10^2$ |
| # blocks | 6 | 7 | 8 | 9 | 10 |
| # states | 4051 | 37633 | 394353 | 4596553 | 58941091 |
| approx. | $4.1 \times 10^3$ | $3.8 \times 10^4$ | $3.9 \times 10^5$ | $4.6 \times 10^6$ | $5.9 \times 10^7$ |
| #blocks | 11 | 12 | 13 | 14 | 15 |
| # states | 824073141 | 12470162233 | 202976401213 | 3535017524403 | 65573803186921 |
| approx. | $8.2 \times 10^8$ | $1.3 \times 10^{10}$ | $2.0 \times 10^{11}$ | $3.5 \times 10^{12}$ | $6.6 \times 10^{13}$ |
| #blocks | 16 | ... | | | |
| # states | 1290434218669921 | ... | | | |
| approx. | $1.3 \times 10^{15}$ | | | | |

The search tree might be even larger than the number of legal states of a problem – i. e., the number of nodes in the state-space. First, some states can be constructed more than once (if cycle detection is restricted to paths) and second, for some planning algorithms, "illegal" states which do not belong to the state space might be constructed. Backward planning by goal regression can lead to such illegal states (see sect. 3.4).

An analysis of the complexity of Strips planning is given by (Bylander, 1994). Another approach to planning is that the state space (or that part of it which might contain the searched for solution) is already given. The task of the planner is than to extract a solution from the search space (this strategy is used by Graphplan, see sect. 4.2.1). This problem is still NP-complete!

Because planning is an inherently hard problem, no planning approach can outperform alternative approaches in every domain. Instead, different planning approaches are better suited for different kinds of domains.

### 3.3.2 TERMINATION, SOUNDNESS, COMPLETENESS

A planning algorithm should – like every search algorithm – be correct and complete:

**Definition 2.10 (Soundness)** *A planning algorithm is sound if it only generates legal solutions for a planning problem. That is, the generated sequence of actions transforms the given initial state into a state satisfying the given top-level goals. Soundness implies that the generated plans are consistent: A state generated by applying an action to a previous state is consistent if it does not contain contradictory literals, i. e., if it belongs to the domain. A solution is consistent if it does not contain contradictions with regard to variable bindings and to ordering of states.*

Proof of soundness relies on the operational semantics given for operator application (such as our definitions 2.6 and 2.7). The proof can be performed by induction over the length of the sequences of actions.

Correctness follows from soundness and termination:

**Definition 2.11 (Correctness)** *A search algorithm is correct, if it is sound and termination is guaranteed.*

To prove termination, it has to be shown, that for each plan step the search space is reduced such that the termination conditions given for the planning algorithm are eventually reached. For finite domains and a cycle-test covering the search tree, the planner always terminates when the complete search tree was constructed.

Besides making sure that a planner always terminates and that if returns a plan that plan is a solution to the input problem, it is desirable that the planner returns a solution for all problems, where a solution exists:

**Definition 2.12 (Completeness)** *A search algorithm is complete, if it finds a solution if such a solution exists. That is, if the algorithm terminates without a solution, then the planning problem has no solution.*

To proof completeness, it has to be shown that the planner only then terminates without a solution, if no solution exists for a problem.

We will give proofs for correctness and completeness for our planner DPlan in chapter 3.

Backward planning algorithms based on a linear strategy are incomplete. The Sussman anomaly is based on that incompleteness. Incompleteness of linear backward planning is discussed in section 3.4.2.

### 3.3.3 OPTIMALITY

When abstracting from different costs (such as time, energy use) of operator application, optimality of a plan is defined with respect to its length. Operator applications are assumed to have uniform costs (for example one unit per application) and the cost of a plan is equivalent to the number of actions it contains.

**Definition 2.13 (Optimality of a Plan)** *A plan is optimal if each other plan which is a solution for the given problem has equal or greater length.*

For operators involving different (positive) costs a plan is optimal if for each other plan which is a solution the sum of the costs of the actions in the plan is equal or higher.

For uninformed planning based on depth-first search, as described in section 3.2, optimality of plans cannot be guaranteed. In fact, there is a trade-off

between efficiency and optimality – to generate optimal plans, the state-space has to be searched more exhaustively. The obvious search strategy for obtaining optimal plans is breadth-first search.[10] Universal planning (see sect. 4.4) for deterministic domains results in optimal plans.

## 3.4 BACKWARD PLANNING

Backward planning often results in smaller search trees than forward planning because the top-level goals can be used to guide the search. For a long time planning was used as synonym for backward planning while forward planning was often associated with problem solving (see sect. 4.5.1). For backward planning, plan construction starts with the top-level goals and in each planning step a *predecessor* state is generated by backward operator application. Backward planning is also called *regression planning*.

Before we go into the details of backward planning, we present the backward search tree for the tree for the example we already presented for forward search (see fig. 2.8). The backward search tree is slightly smaller than the forward search tree. Ignoring the states which were generated as backtrack-points, in forward search thirteen states have to be visited, in backward-search nine. In general, the savings can be much higher.

There is a price to pay for obtaining smaller search trees. The problem is, that a backward operator application can produce an inconsistent state description (see def. 2.12). We will discuss this problem in section 3 in chapter 3, in the context of our state-based non-linear backward planner DPlan. In the following, we will discuss the classic (Strips) approach to backward planning, using goal regression together with the problem of detecting and eliminating inconsistent state descriptions. Furthermore, we will address the problem of incompleteness of classical linear backward planning and present non-linear planning, which is complete.

### 3.4.1 GOAL REGRESSION

Regression planning starts with the top-level goals of a planning problem as input (root of the search tree). For a planning problem involving three blocks $A$, $B$, and $C$ and the top-level goals $\mathcal{G}$ = {*on(A, B), on(B, C)*}, the state depicted in the root node in figure 2.8 is the only legal goal state with $\mathcal{G} \subseteq s$ = {*on(A, B), on(B, C), ontable(C), clear(A)*}. In contrast to forward planning, planning does not start with a complete state description $s$ but with a partial state description $\mathcal{G}$ A planning step is realized by goal regression:

[10] Alternatively depth-first search can be extended such that all plans are generated – always keeping the current shortest plan.

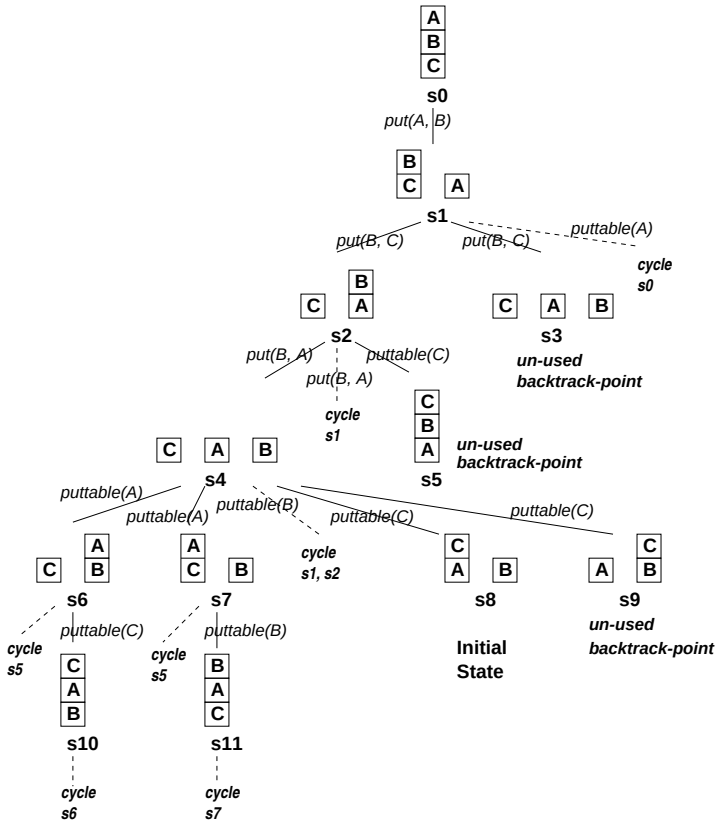



*Figure 2.8.* A Backward Search Tree for Blocks-World

**Definition 2.14 (Goal Regression)** *Let $G$ be a set of (goal) literals and $o$ an instantiated operator.*

- *If for an atom $p \in G$ holds $p \in ADD(o)$, then the goal corresponding to $p$ can be replaced by* true *which is equivalent to removing $p$ from $G$.*
- *If for an atom $p \in G$ holds $p \in DEL(o)$, then the goal corresponding to $p$ is destroyed, that is, it must be replaced by* false *which is equivalent to replacing formula $G$ by* false*.*
- *If $p \in G$ is neither contained in ADD(o) nor in DEL(o), nothing is changed.*

If a formula is reduced to *false*, an inconsistent state description is detected. Another possibility to deal with inconsistency is to introduce domain axioms and deduce contradictions by theorem proving. We give the beginning of the search tree using goal regression in figure 2.9. A complete regression tree

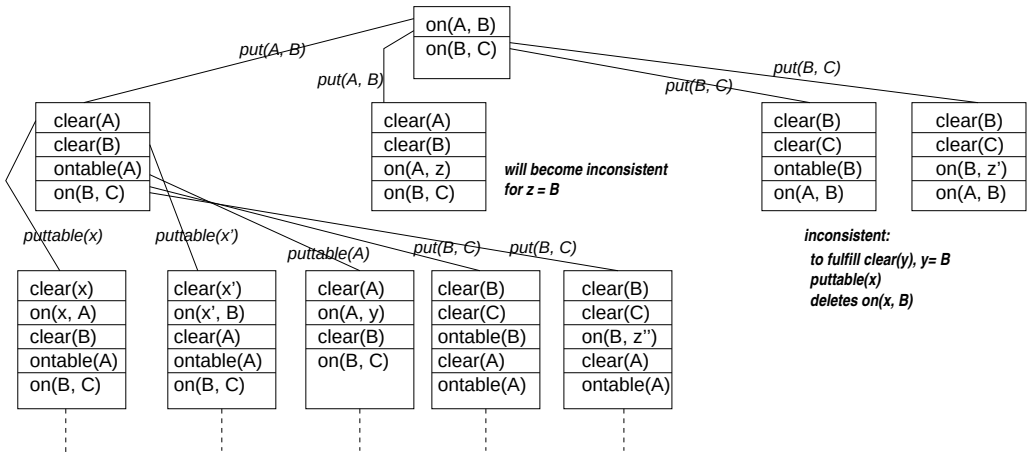



*Figure 2.9.* Goal-Regression for Blocks-World

using the domain specification given in figure 2.4 is given in (Nilsson, 1980, pp. 293–295). Using goal regression, there can occur free variables when introducing new subgoals. Typically, these variables can be instantiated if a subgoal expression is reached which matches with the initial state. Depending on the domain, there might go a lot of effort in dealing with inconsistent states. If more complex operator specifications – such as conditional effects and all-quantified expressions – are allowed, the number of impossible states which are constructed and must be detected during plan construction can explode.

To summarize, there are the following differences between forward and backward planning:

**Complete vs. Partial State Descriptions.** Forward planning starts with a complete state representation – the initial state – while backward planning starts with a partial state representation – the top-level goals (which might contain variables).

**Consistency of State Descriptions.** A planning step in forward planning always generates a consistent successor state. Soundness of forward planning follows easily from the soundness of the planning steps. A planning step in backward planning can result in an inconsistent state description. In general, a planner might not detect all inconsistencies. To proof soundness, it must be shown, that if a plan is returned, all intermediate state representations on the path from the top-level goals to the initial state are consistent.

**Variable Bindings.** In forward planning, all constructed state representations are fully instantiated. This is due to the way, in which planning operators

are defined: Usually, all variables occurring in an operator are bound by the precondition. In backward planning, newly introduced subgoals (i. e., preconditions of an operator) might contain variables which are not bound by matching the current subgoals with an operator.

In chapter 3 we will introduce a backward planning strategy based on complete state descriptions: Planning starts with a goal *state* instead with the top-level goals. An operator is applicable if *all* elements of its ADD-list match with the current state. Operator application is performed by removing all elements of the ADD-list from the current descriptions, i. e., all literals are reduced to true in one step, and by adding the union of preconditions and DEL-list (see def. 2.7). Since usually the elements of the DEL-list are a subset of the elements of the preconditions, this state-based backward operator application can be seen as a special kind of goal regression.

A further difference between forward and backward planning is:

**Goal Ordering.** In forward planning, it must be decided which of the action candidates whose preconditions are satisfied in the current state is applied. In backward planning, it must be decided which (sub-)goal of a list of goals is considered next. That is, backward planning involves goal ordering.

In the following, we will show that the original linear strategy for backward planning is incomplete.

### 3.4.2 INCOMPLETENESS OF LINEAR PLANNING

A planning strategy is called linear if it does not allow interleaving of sub-goals. That means, plan construction is based on the assumption that a problem can be solved by solving each goal separately (Sacerdoti, 1975). This assumption does only hold for independent goals – Nilsson (1980) uses the term “commutativity”, (Georgeff, 1987, see also).

A famous demonstration for incompleteness of linear backward planning is the Sussman Anomaly (see Waldinger, 1977, for a discussion) illustrated in figure 2.10. Here, the linear strategy does not work, regardless in which order the sub-goals are approached. If $B$ is put on $C$, $A$ is covered by these two blocks and can only be moved, if the goal *on(B, C)* is destroyed. For reaching the goal *on(A, B)*, first $A$ has to be cleared by putting $C$ on the table, if $A$ is subsequently put on $B$, $C$ cannot be moved under $B$ without destroying goal *on(A, B)*.

Linear planning corresponds to dealing with goals organized in a *stack*:

*[on(A, B), on(B, C)]*

try to satisfy goal *on(A, B)*

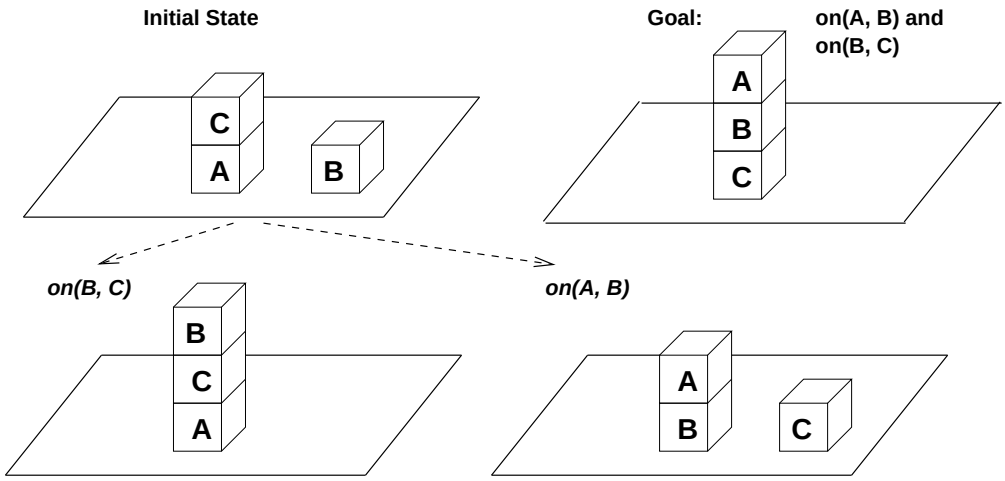



*Figure 2.10.* The Sussman Anomaly

solve sub-goals *[clear(A), clear(B)]*[11]

all sub-goals hold after *puttable(C)*

apply *put(A, B)*

goal *on(A, B)* is reached

try to satisfy goal *on(B, C)*.

Interleaving of goals – also called *non-linear* planning – allows that a sequence of planning steps dealing with one goal is interrupted to deal with another goal. For the Sussman Anomaly, that means that after block $C$ is put on the table pursuing goal *on(A, B)*, the planner switches to the goal *on(B, C)*. Non-linear planning corresponds to dealing with goals organized in a *set*:

{*on(A, B), on(B, C)*}

try to satisfy goal *on(A, B)*

{*clear(A), clear(B), on(A, B), on(B, C)*}

*clear(A)* and *clear(B)* hold after *puttable(C)*

try to satisfy goal *on(B, C)*

apply *put(B, C)*

[11] We ignore the additional subgoal *ontable(A)* rsp. *on(A, z)* here.

try to satisfy goal *on(A, B)*

apply *put(A, B)*.

The correct sequence of goals might not be found immediately but involve backtracking.

Another example to illustrate the incompleteness of linear planning was presented by Veloso and Carbonell (1993): Given are a rocket and several packages together with operators for loading and unloading packages in and from the rocket and an operator for shooting the rocket to the moon – but no operator for driving the rocket back from the moon. The planning goal is to transport some packages, for example *at(PackA, Moon), at(PackB, Moon), at(PackC, Moon)*. If the goals are addressed in a linear way, one package would be loaded in the rocket, the rocket would go to the moon, and the package would be unloaded. The rest of the packages could never be delivered! The correct plan for this problem is, to load *all* packages in the rocket before the rocket moves to its destination. We will give a plan for the rocket domain in chapter 3 and we will show in chapter 8 how this strategy for solving problems of the rocket domain can be learned from some initial planning experience.

A second source of incompleteness is, if a planner instantiates variables in an eager way – also called strong commitment planning. An illustration with a register-swapping problem is given in (pp. 305–307 Nilsson, 1980). A similar problem – sorting of arrays – and its solution is discussed in (Waldinger, 1977). We will introduce sorting problems in chapter 3. Modern planners are based on a non-linear, least commitment strategy.

# 4 PLANNING SYSTEMS

In this section we give a short overview of the history of and recent trends in planning research. We will only give more detailled descriptions for such research areas which are relevant for the later parts of the book. Otherwise, we will only give short characterizations together with hints to the literature.

## 4.1 CLASSICAL APPROACHES

### 4.1.1 STRIPS PLANNING

The first well-known planning system was Strips (Fikes and Nilsson, 1971). It integrated concepts developed in the area of problem solving by state-space search and means-end analysis – as realized in the General Problem Solver (GPS) from Newell and Simon (1961) (see sect. 4.5.1) – and concepts from theorem proving and situation calculus – the QA3 system of Green (1969). As discussed above, the basic notions of the Strips language are still the core of modern planning languages, as for example PDDL. The Strips planning algorithm – based on goal regression and linear planning, as described above

–, on the other hand, was replaced end of the eighties by non-linear, least-commitment approaches. In the seventies and eighties, lots of work addressed problems with the original Strips approach, for example (Waldinger, 1977; Lifschitz, 1987).

### 4.1.2 DEDUCTIVE PLANNING

The deductive approach to planning as theorem proving introduced by Green was pursued through the seventies and eighties mainly by Manna and Waldinger (Manna and Waldinger, 1987). Manna and Waldinger combined deductive planning and deductive program synthesis, and we will discuss their work in chapter 6. Current deductive approaches are based on so called *action languages* where actions are represented as temporal logic formulas. Here plan construction is a process of reasoning about change (Gelfond and Lifschitz, 1993). End of the nineties, symbolic model checking was introduced as an approach to planning as verification of temporal formulas in an semantic model (Giunchiglia and Traverso, 1999). Symbolic model checking is mainly applied in universal planning for deterministic and non-deterministic domains (see sect. 4.4).

### 4.1.3 PARTIAL ORDER PLANNING

Also in the seventies, the first partial order planner (NOAH) was presented by Sacerdoti (1975).[12] Partial order planning is based on a search in the space of (incomplete) plans. Search starts with a plan containing only the initial state and the top-level goals. In each planning step, the plan is *refined* by either introducing an action fulfilling a goal or a precondition of another action, or by introducing an ordering that puts one action in front of another action, or by instantiating a previously unbound variable. Partial order planners are based on a non-linear strategy – in each planning step an arbitrary goal or precondition can be focussed. Furthermore, the least commitment strategy – i. e., refraining from committing to a specific ordering of planning steps or to a specific instantiation of a variable as long as possible – was introduced in the context of partial order planning (Penberthy and Weld, 1992).

The resulting plan is usually not a totally, but only a partially ordered set of actions. Actions for which no ordering constraints occurred during plan construction remain unordered. A totally ordered plan can be extracted from the partially ordered plan by putting parallel (i. e., independent) steps in an arbitrary order. An overview of partial order planning together with a survey of the most important contributions to this research is given in (Russell and

[12] Sacerdoti called NOAH an hierarchical planner. Today, hierarchical planning means that a plan is constructed on different levels of abstraction (see sect. 4.3.1).

Norvig, 1995, chap. 11). Partial order planning was the dominant approach to planning from end of the eighties to mid of the nineties. The main contribution of partial order planning is the introduction of non-linear planning and least commitment.

### 4.1.4 TOTAL ORDER NON-LINEAR PLANNING

Also at the end of the eighties, the Prodigy system was introduced (Veloso, Carbonell, Pérez, Borrajo, Fink, and Blythe, 1995). Prodigy is a state-based, total-order planner based on a non-linear strategy. Prodigy is more a framework for planning than a single planning system. It allows the selection of a variety of planning strategies and, more important, it allows that search is guided by domain-specific control knowledge – turning a domain-independent into a more efficient domain-specific planner. Prodigy includes different strategies for learning such control knowledge from experience (see sect. 5.2 and offers techniques for reusing already constructed plans in the context of new problems based on analogical reasoning.

Two other total-order, non-linear backward planners are HSPr (Haslum and Geffner, 2000) and DPlan (Schmid and Wysotzki, 2000b) – which we will present in detail in chapter 3.

## 4.2 CURRENT APPROACHES

### 4.2.1 GRAPHPLAN AND DERIVATES

Since the mid of the nineties, a new, more efficient, generation of planning algorithms, dominates the field. The *Graphplan* approach presented by Blum and Furst (1997) can be seen as the starting point of the new development. Graphplan deals with planning as network flow problem (Cormen et al., 1990). The process of plan construction is divided into two parts: first, a so called *planning graph* is constructed by forward search, second a partial order plan is extracted by backward search. Plan extraction from a planning graph of fixed sized corresponds to a bounded-length plan construction problem.

The planning graph can be seen as a partial representation of the state-space of a problem, representing a sub-graph of the state-space which contains paths from the given initial state to the given planning goals. An example for a plannign graph is given in figure 2.11. It contains fully instantiated literals and actions. But, in contrast to a state-space representation, nodes in the graph are not states – i. e., conjunctions of literals – but single literals (propositions). Because the number of different literals in a domain is considerably smaller than the number of different sub-sets of those literals (i. e., state representations), the size of a planning graph does not grow exponentially (see sect. 3.3.1).

The planning graph is organized level-wise. The first level is the set of propositions contained in the initial state, the next level is a set of actions, a

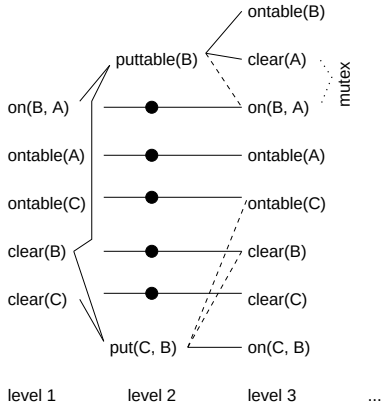



*Figure 2.11.* Part of a Planning Graph as Constructed by Graphplan

proposition from the first level is connected with an action in the second level, if it is a precondition for this action. The next level contains propositions again. For each action of the preceding level, all propositions contained in its ADD-list are introduced (and connected with the action). The planning graph contains so called *noop* actions at each level which just pass a literal from level $k$ to level $k+2$. Furthermore, so called *mutex* relations are introduced at each level, representing (an incomplete set) of propositions or actions which are mutually exclusive on a given level. Two actions are mutex if they interfere (one action deletes a precondition or ADD-effect of the other) or have competing needs (have mutually exclusive preconditions). Two propositions $p$ and $q$ are mutex if each action having an add-edge to proposition $p$ is marked as mutex of each action having an add-edge to proposition $q$. Construction of a planning graph terminates, when the first time a level contains all literals occurring in the planning goal and these literals are not mutex. If backward plan extraction fails, the planning graph is extended one level.

Originally, Graphplan was developed for the Strips planning language. Koehler et al. (1997) presented an extension to conditional and universally quantified operator effects (system IPP). Another successful Graphplan based system is STAN (Long and Fox, 1999). After Graphplan, a variety of so called compilation approaches have become popular. Starting with a planning graph, the bounded-length plan construction problem is addressed by different approaches to solving canonical combinatorial problems, such as satisfiability-solvers (Kautz and Selman, 1996, Blackbox), integer programming, or constraint satisfaction algorithms (see Kambhampati, 2000, for a survey). IPP, STAN, and Blackbox solve blocks-world problems up to 10 objects in under a second and up to thirteen objects in under 1000 seconds (Bacchus et al., 2000).

Traditional planning algorithms worked for a (small) sub-set of first-order logic where in each planning step variables occurring in the operators must be instantiated. In the context of partial order planning, the expressive power of the original Strips approach was extended by concepts included in the PDDL language as discussed in section 2.1 (Penberthy and Weld, 1992). In contrast, compilation approaches are based on propositional logic. The reduction in the expressiveness of the underlying language gives rise to a gain in efficiency for plan construction. Another approach based on propositional representations is symbolic model checking which we will discuss in context with universal planning (see sect. 4.4).

### 4.2.2 FORWARD PLANNING REVISITED

Another efficient planning system of the late nineties, is the system HSP from Bonet and Geffner (1999). HSP is a forward planner based on heuristic search. The power of HSP is based on an efficient procedure for calculating a lower bound estimate $h'(s)$ for the distance (remaining number of actions) from the current state $s$ to a goal state. The heuristic function $h'(s)$ is calculated for a "relaxed" problem $P'$, which is obtained from the given planning problem $P$ by ignoring the DEL-lists of the operators. Thus, $h'(s)$ is set to the number of actions which transform $s$ in a state where the goal literals appear for the first time – ignoring that preconditions of these actions might be deleted on the way.

The new generation of forward planners dominated in the 2000 AIPS competition (Bacchus et al., 2000). For example, HSP can solve blocks-world problems up to 35 blocks in under 1000 seconds.

## 4.3 COMPLEX DOMAINS AND UNCERTAIN ENVIRONMENTS

### 4.3.1 INCLUDING DOMAIN KNOWLEDGE

Most realistic domains are by far more complex than the blocks-world domain discussed so far. Domain specifications for more realistic domains – such as assembling of machines or logistics problems – might involve large sets of (primitive) operators and/or huge state-spaces. The obvious approach to generate plans for complex domains is to restrict search by providing domain-specific knowledge. Planners relying on domain-specific knowledge can solve blocks-world problems with up to 95 blocks under one second (Bacchus et al., 2000). In contrast to the *general purpose planners* discussed so far, such knowledge-based systems are called *special purpose planners*. Note that all domain-specific approaches require that more effort and time is invested in the development of a formalized domain model. Often, such knowledge is not easily to provide.

One way to deal with a complex domain, is to specify plans at different levels of detail. For example (see Russell and Norvig, 1995, p. 368), to launch a rocket, a top-level plan might be: prepare booster rocket, prepare capsule, load cargo, launch. This plan can be differentiated to several intermediate-level plans until a plan containing executable actions is reached (on the detail of insert nut $A$ into hole $B$, etc.). This approach is called *hierarchical planning*. For hierarchical planning, additional to primitive operators which can be instantiated to executable actions, abstract operators must be specified. An abstract operator represents a decomposition of a problem into smaller problems. To specify an abstract operator, knowledge about the structure of a domain is necessary. An introduction to hierarchical planning by problem decomposition is given in (Russell and Norvig, 1995, chap. 12).

Other techniques to make planning feasible for complex domains are concerned with reducing the effort of search. One source of complexity is that meaningless instantiations and inconsistent states might be generated which must be recognized and removed (see discussion in sect. 3.4). This problem can be reduced or eliminated if domain knowledge is provided in the form of axioms and types. A second source of complexity is that uninformed search might lead into areas of the state-space which are far away from a possible solution. This problem can be reduced by providing domain specific control strategies which guide search. Planners that rely on such domain-specific control knowledge can easily outperform every domain-independent planner (Bacchus and Kabanza, 1996).

Alternatively to explicitly providing a planner with such kind of domain-specific information, this information can be obtained by extracting information from domain specifications by pre-planning analysis (see sect. 5.1) or from some example plans using machine learning techniques (see sect. 5.2).

### 4.3.2 PLANNING FOR NON-DETERMINISTIC DOMAINS

Constructing plans for real-world domains must take into account that information might be incomplete or incorrect. For example, it might be unknown at planning time, whether the weather conditions are sunny or rainy when the rocket is to be launched (see above); or during planning time it is assumed that a certain tool is stored in a certain shelf, but at plan execution time, the tool might have been moved to another location. The first example addresses incompleteness of information due to environmental changes. The second example addresses incorrect information which might be due to environmental changes (e.g. , an agent which does not correspond to the agent executing the plan moved the tool) or to non-deterministic actions (e.g. , depending on where the planning agent moves after using the tool, he might place the tool back in the shelf or not).

A classical approach to deal with incomplete information is *conditional planning*. A conditional plan is a disjunction of sub-plans for different contexts (as good or bad weather conditions). We will see in chapter 8 that introducing conditions into a plan is a necessary step for combining planning and program synthesis. A classical approach to deal with incorrect information is *execution monitoring and re-planning*. Plan execution is monitored and if a violation of the preconditions for the action which should be executed next is detected, re-planning is invoked. An introduction to both techniques is given in (Russell and Norvig, 1995, chap. 13).

Another approach to deal with non-deterministic domains is *reactive planning*. Here, a set or table of state-action rules is generated. Instead of executing a complete plan, the current state of the environment triggers which action is performed next, dependent on what conditions are fulfilled in the current state. This approach is also called *policy learning* and is extensively researched in the domain of reinforcement learning (Dean, Basye, and Shewchuk, 1993; Sutton and Barto, 1998). A similar approach, combining problem solving and decision tree learning, is proposed by (Müller and Wysotzki, 1995). In the context of symbolic planning, *universal planning* was proposed as an approach to policy learning (see sect. 4.4).

## 4.4 UNIVERSAL PLANNING

Universal planning was originally proposed by Schoppers (1987) as an approach to learn state-action rules for non-deterministic domains. A universal plan represents solution paths for all possible states of a planning problem, instead of a solution for one single initial state. State-action rules are extracted from a universal plan covering all possible states of a given planning problem. Generating universal plans instead of plans for a single initial state was also proposed by Wysotzki (1987) in the context of inductive program synthesis (see part II).

A universal plan corresponds to a breadth-first search tree. Search is performed backward, starting with the top-level goals and for each node at the current level of the plan all (new and consistent) predecessor nodes are generated. The set of predecessor nodes of a set of nodes $S$ is also called pre-image of $S$. An abstract algorithm for universal plan construction is given in table 2.4. Universal planning was criticized as impracticable (Ginsberg, 1989), because such search trees can grow exponentially (see discussion of PSPACE-completeness, sect. 3.3.1). Currently, universal planning has a renaissance, due to the introduction of OBDDs (ordered binary decision diagrams) as a method for a compact representation of universal plans. OBDD-representations were originally developed in the context of hardware design (Bryant, 1986) and later adopted in symbolic model checking for efficient exploration of large state-spaces (Burch, Clarke, McMillan, and Hwang, 1992). The new, memory

*Table 2.4.* Planning as Model Checking Algorithm (Giunchiglia, 1999, fig. 4)

**function** PLAN(P) where *P(D, I, G)* is a planning problem with

- $I = \{s_0\}$ as initial state,
- $G$ as goals, and
- $D = \langle F, S, A, R \rangle$ as planning domain with $F$ as set of fluents, $S \subseteq 2^F$ as finite set of states, $A$ as finite set of actions, and $R : S \times A \rightarrow S$ as transition function. An action $a \in A$ is executable in $s \in S$ if $R(s, a) \neq \emptyset$.

CurrentStates := $\emptyset$; NextStates := $G$; Plan := $\emptyset$;

**while** (NextStates $\neq$ CurrentStates) **do** (*)

**if** I $\subseteq$ NextStates **then return** Plan; (**)

OneStepPlan := ONESTEPPLAN(NextStates,D);
*(calculate pre-image of NextStates)*

Plan := Plan $\cup$ PRUNESTATES(OneStepPlan,NextStates);
*(eliminate states which have already been visited)*

CurrentStates := NextStates;

NextStates := NextStates $\cup$ PROJECTACTIONS(OneStepPlan);
*(*PROJECTACTIONS*, given a set of state-action pairs, returns the corresponding set of states)*

**return** Fail.

efficient approaches to universal planning are based on the idea of viewing planning as model checking paradigm instead of planning as a state-space search problem (like Strips planners) or as theorem proving problem (like deductive planners). Planning as model checking is not only successfully applied to non-deterministic domains – planners MBP(Cimatti, Roveri, and Traverso, 1998) and UMOP (Jensen and Veloso, 2000) –, but also to benchmark problems for planning in deterministic domains (Edelkamp, 2000, planner MIPS). Because plan construction is based on breadth-first search, universal plans for deterministic domains represent optimal solutions.

The general idea of planning as model checking is that a planning domain is described as semantic model of a domain. A semantic model can be given for example as a finite state machine (Cimatti et al., 1998) or as Kripke structure (Giunchiglia and Traverso, 1999). A domain model $D$ represents the states and actions of the domain and the state transitions caused by the execution of actions. States are represented as conjunctions of atoms. State transitions *(state, action, state')* can be represented as formulas $state \wedge action \rightarrow state'$. As usual, a planning problem is the problem of finding plans of actions given planning domain, initial and goal states. Plan generation is done by exploring

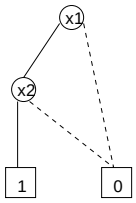



*Figure 2.12.* Representation of the Boolean Formula $f(x_1, x_2) = x_1 \wedge x_2$ as OBDD

the state space of the semantic model. At each step, plans are generated by checking the truth of some formulas in the model. Plans are also represented as formulas and planning is modeled as search through sets of states (instead of single states) by evaluating the assignments verifying the corresponding formulas. Giunchiglia and Traverso (1999) gives an overview of planning as model checking.

As mentioned above, plans (as semantic models) can be represented compactly as OBDDs. An OBDD is a canonical representation of a boolean function as a DAG (directed acyclic graph). An example for an OBDD representation is given in figure 2.12. Solid lines represent that the preceding variable is true, broken lines represent that it is false. Note, that the size of an OBDD is highly dependent on the ordering of variables (Bryant, 1986).

Our planning system DPlan is a universal planner for deterministic domains (see chapt. 3). In contrast to the algorithm given in table 2.4, planning problems in DPlan only give planning goals, but not an initial state. DPlan terminates, if all states which are reachable from the top-level goals (by calculating the preimages) are enumerated. That is, DPlan terminates successfully, if condition (*) given in table 2.4 is no longer fulfilled and DPlan does not include condition (**). Furthermore, in DPlan the universal plan is represented as DAG over states and not as OBDD. Therefore, DPlan is memory inefficient. But, as we will discuss in later chapters, DPlan is typically applied only to planning problems with small complexity (involving not more than three or four objects). The universal plan is used as starting point for inducing a domain specific control program for generating (optimal) action sequences for problems with arbitrary complexity (see sect.5.2).

## 4.5 PLANNING AND RELATED FIELDS

### 4.5.1 PLANNING AND PROBLEM SOLVING

The distinction between planning and problem solving is not very clear-cut. In the following, we give some discriminations found in the literature (Russell and Norvig, 1995, e. g.,): Often, forward search based complete state representations is classified as problem solving, while backward search based

on incomplete state representations is classified as planning. While planning is based on a logical representation of states, problem solving can rely on different, special purpose representations, for example feature vectors. While planning is mostly associated with a domain-independent approach, problem solving typically relies on pre-defined domain specific knowledge. Examples are heuristic functions as used to guide $A^*$ search (Nilsson, 1971, 1980) or the difference table used in GPS (Newell and Simon, 1961; Nilsson, 1980).

Cognitive psychology is typically concerned with human problem solving and not with human planning. Computer models of human problem solving are mostly realized as production systems (Mayer, 1983; Anderson, 1995). A production system consists of an interpreter and a set of production rules. The interpreter realizes the match-select-apply cycles. A production rule is an IF-THEN-rule. A rule fires if the condition specified in its if-part is satisfied in a current state in working memory. Selection might be influenced by the number of times a rule was already applied successfully – coded as a strength value. Application of a production rule results in a state change. Production rules are similar to operators. But while an operator is specified by preconditions and effects represented as sets of literals, production rules can encode conditions and effects differently. For example, a condition might represent a sequence of symbols which must occur in the current state and the then-part on the rule might give a sequence of symbols which are appended to the current state representation. In cognitive science, production systems are usually goal-directed – the if-part of a rule represents a currently open goal and the then-part either introduces new sub-goals or specifies an action. Goal-driven production systems are similar to hierarchical planners.

The earliest and most influential approach to problem solving in AI and cognitive psychology is the General Problem Solver (GPS) from (Newell and Simon, 1961). GPS is very similar to Strips planning. The main difference is, that selection of a rule (operator) is guided by a difference table. The difference table has to be provided explicitly for each application domain. For each rule, it is represented which difference between a current state and a goal state it can reduce. For example, a rule for *put(A, B)* fulfills the goal *on(A, B)*. The process of identifying differences and selecting the appropriate rule is called means-end analysis. The system starts with the top-level goals and a current state. It selects one of the top-level goals, if this goal is already satisfied in the current state, the system proceeds with the next goal. Otherwise, a rule is selected from the difference table. The top-level goal is removed from the list (stack) of goals and replaced by the preconditions of the rule and so on. As Strips, GPS is based on a linear strategy and is therefore incomplete. While incompleteness is not a desirable property for an AI program, it might be appropriate to characterize human problem solving. A human problem solver usually wants to generate a solution within reasonable time-bounds and can

furthermore only hold a restricted amount of information in his/her short-term memory. Therefore, using an incomplete but simple strategy is rational because it still can work for a large class of problems occurring in everyday life (Simon, 1958).

### 4.5.2 PLANNING AND SCHEDULING

Planning deals with finding such activities which must be performed to satisfy a given goal. Scheduling deals with finding an (optimal) allocation of activities (jobs) to time segments given limited resources (Zweben and Fox, 1994). For a long time planning and scheduling research were completely separated. Only since the last years, the interests of both communities converge. While scheduling systems are successfully applied in many areas (such as factories and transportation companies), planning is mostly done "by hand". Currently, researchers and companies become more interested in automatizing the planning part. This goal seems now far more realistic than five years ago, due to the emergence of new, efficient planning algorithms. On the other hand, researchers in planning aim at applying their approaches at realistic domains – such as the logistics domain and the elevator domain used as benchmark problems in the AIPS-00 planning competition (Bacchus et al., 2000).[13] Planning in realistic domains often involves dealing with resource constraints. One approach, for example proposed by Do and Kambhampati (2000), is to model planning and scheduling as two succeeding phases, both solved with a constraint satisfaction approach. Other work aims at integrating resource constraints in plan construction (Koehler, 1998; Geffner, 2000). In chapter 4 we present an approach to integrating function application in planning and example applications to planning with resource constraints.

### 4.5.3 PROOF PLANNING

In the domain of automatic theorem proving, planning is applied to guide proof construction. Proof planning was originally proposed by Bundy (1988). Planning operators represent *proof methods* with preconditions, postconditions, and tactics. A tactic represents a number of inference steps. It can be applied if the preconditions hold and the postconditions must be guaranteed to hold after the inference steps are executed. A tactic guides the search for a proof by prescribing that certain inference steps should be executed in a certain sequence given some current step in a proof.

An example for a high-level proof method is "proof by mathematical induction". Bundy (1988) introduces a heuristics called "rippling" to describe

[13] Because of the converging interests in planning and scheduling research, the conference Artificial Intelligence Planning Systems was renamed into Artificial Intelligence Planning and Scheduling.

the heuristics underlying the Boyer-Moore theorem prover and represents this heuristics as a method. Such a method specifies tactics on different levels of detail – similar to abstract and primitive operators in hierarchical planning. Thus, a "super-method" for guiding the construction of a complete proof invokes several sub-methods which satisfy the post-condition of the super-method after their execution.

Proof planning, like hierarchical planning, requires insight in the structure of the planning domain. Identifying and formalizing such knowledge can be very time-consuming, error-prone, or even impossible, as discussed in section 4.3.1. Again, learning might be a good alternative to explicitly specifying such domain-dependent knowledge: First, proofs are generated using only primitive tactics. Such proofs can then be input in a machine learning algorithm and generalized to more general methods (Jamnik, Kerber, and Benzmüller, 2000). In principle, all strategies available for learning control rules or control programs for planning – as discussed in section 5.2 – can be applied to proof planning. Of course, such a learned method might be incomplete or non-optimal because it is induced from some specific examples. But the same is true for methods which are generated "by hand". In both cases, such strategic decisions should not be executed automatically. Instead, they can be offered to a system user and accepted or reject by user interaction.

## 4.6 PLANNING LITERATURE

Classical Strips planning is described in all introductory AI textbooks – such as Nilsson (1980) and Winston (1992). The newest textbook from (Russell and Norvig, 1995) focusses on partial order planning, which is also described in Winston (1992). Short introductions to situation calculus can also be found in all textbooks. A collection of influential papers in planning research from the beginning until the end of the eighties is presented by Allen, Hendler, and (Eds.) (1990).

The area of planning research underwent significant changes since the mid-nineties. The newer Graphplan based and compilation approaches as well as methods of domain-knowledge learning (see sect. 5) are not described in textbooks. Good sources to obtain an overview of current research are the proceedings of the *AIPS conference* (Artificial Intelligence Planning and Scheduling; `http://www-aig.jpl.nasa.gov/public/aips00/`) and the *Journal of Artificial Intelligence Research* (JAIR; `http://www.cs.washington.edu/research/jair/home.html`). Overview papers of special areas of planning can be found in the *AI Magazine* – for example, a recent overview of current planning approaches by Weld (1999). Furthermore, research in planning is presented in all major AI conferences (IJCAI, AAAI) and AI journals (e.g. , *Artificial Intelligence*). A collection of current planning systems together

with the possibility to execute planners on a variety of problems can be found at `http://rukbat.fokus.gmd.de:8080/`.[14]

# 5 AUTOMATIC KNOWLEDGE ACQUISITION FOR PLANNING

## 5.1 PRE-PLANNING ANALYSIS

Pre-planning analysis was proposed for example by (Fox and Long, 1998) to eliminate meaningless instantiations when construction a propositional representation of a planning problem, as for example a planning graph. The basic idea is, to automatically infer types from domain specifications. These types then are used to extract state invariants. Additionally to eliminating meaningless instantiations state invariants can be used to detect and eliminate unsound plans. The domain analysis of the system TIM proposed by Fox and Long (1998) extracts information by analyzing the literals occurring in the preconditions, ADD- and DEL-lists of operators (transforming them in a set of finite state machines).

For example, for the *rocket* domain (shortly described in sect. 3.4.2), it can be inferred, that each package is always either in the rocket or at a fixed place, but never at two locations simultaneously.

Extracting knowledge from domain specifications makes planning more efficient because it reduces the search space by eliminating impossible states. Another possibility to make plan construction more efficient is to guide search by introducing domain specific control knowledge.

## 5.2 PLANNING AND LEARNING

Enriching domains with control knowledge which guides a planner to reduce search for a solution makes planning more efficient and therefore makes a larger class of problems solvable under given time and memory restrictions. Because it is not always easy to provide such knowledge and because domain modeling is an “art” which is time consuming and error-prone, one area of planning research deals with the development of approaches to learn such knowledge automatically from some sample experience with a domain.

### 5.2.1 LINEAR MACRO-OPERATORS

In the eighties, approaches to *learning (linear) macro operators* were investigated (Minton, 1985; Korf, 1985): After a plan is constructed for a problem of a given domain, operators which appear directly after each other in the plan can be composed to a single macro operator by merging their preconditions and

[14] This website was realized by Jürgen Müller as part of his diploma thesis at TU Berlin, supervised by Ute Schmid and Fritz Wysotzki.

effects. This process can be applied to pairs or larger sequences of primitive operators. If the planner is confronted with a new problem of the given domain, plan construction might involve a smaller number of match-select-apply cycles because macro operators can be applied which generate larger segments of the searched for plan in one step. This approach to learning in planning, however, did not succeed, mainly because of the so called *utility problem*. If macros are extracted undiscriminated from a plan, the system might become "swamped" with macro-operators. Possible efficiency gains from reduction of the number of match-select-apply cycles are counterbalanced or even overridden by the number of operators which must be matched.

A similar approach to linear macro learning is investigated in the context of cognitive models of human problem solving (see sect. 4.5.1)and learning. The ACT system (Anderson, 1983) and its descendants (Anderson and Lebière, 1998) are realized as production systems with a declarative component representing factual knowledge and a procedural component representing skills. The procedural knowledge is represented by production rules (*if* condition *then* action pairs). Skill acquisition is modelled as "compilation" which includes concatenating primitive rules. This mechanism is used to describe speed-up learning from problem solving experience. Another production system approach to human cognitive skills is the SOAR system (Newell, 1990), a descendant of GPS. Here, "chunking" of rules is invoked, if an impasse during generating a problem solution has occurred and was successfully resolved.

### 5.2.2 LEARNING CONTROL RULES

From the late eighties to the mid of the nineties, control rule learning was mainly investigated in context of the Prodigy system (Veloso et al., 1995). A variety of approaches – mainly based on explanation-based learning (EBL) or generalization (Mitchell, Keller, and Kedar-Cabelli, 1986) – were investigated to learn control rules to improve the efficiency of plan construction and the quality (i. e., optimality) of plans. An overview of all investigated methods is given in Veloso et al. (1995). A control-rule is represented as production rule. For a current state achieved during plan construction, such a control rule provides the planner with information which choice (next action to select, next sub-goal to focus) it should make. The EBL-approach proposed by Minton (1988), extracts such control rules from an analysis of search trees by explaining why certain branching decisions were made during search for a solution.

Another approach to control rule learning, based on learning decision trees (Quinlan, 1986) or decision lists (Rivest, 1987), is closely related to policy learning in reinforcement learning (Sutton and Barto, 1998). For example, Briesemeister, Scheffer, and Wysotzki (1996) combined problem solving with decision tree learning. For a given problem solution, each state is represented as a feature vector and associated with the action which was executed in this

state. A decision tree is induced, representing relevant features of the state description together with the action which has to be performed given specific values of these features. Each path in the decision tree leads to an action (or a "don't know what to do") in its leaf and can be seen as a condition-action rule. After a decision tree is learned, new problem solving episodes can be guided by the information contained in the decision tree. Learning can be performed incrementally, leading either to instantiations of up to now unknown condition-action pairs or to a restructuring of the tree. Similar approaches are proposed by (Martín and Geffner, 2000) and (Huang, Selman, and Kautz, 2000). Martín and Geffner (2000) learn policies represented in a concept language (i. e., as logical formulas) with a decision list approach. They can show that a larger percentage of complex problems (such as blocks-world problems involving 20 blocks) can be successfully solved using the learned policies and that the generated plans are – while not optimal – reasonably short.

### 5.2.3 LEARNING CONTROL PROGRAMS

An alternative approach to learning "molecular" rules is learning of control *programs*. A control program generates a possibly cyclic (recursive, iterative) sequence of actions. We will also speak of (recursive) control rules when refering to control programs. Shell and Carbonell (1989) contrast iterative with linear macros and give a theoretical analysis and some empirical demonstrations of the efficiency gains which can be expected using iterative macros. Iterative macros can be seen as programs because they provide a control structure for repeatedly executing a sequence of actions until the condition for looping does no longer hold. The authors point out that a control program represents strategic knowledge for a domain. Efficient human problem solving should not only be explained by speed-up effects due to operator-merging but also by acquiring problem solving strategies – i. e., knowledge on a higher level of abstraction.

Learning control programs from some initial planning experience brings together inductive program synthesis and planning research which is the main topic of the work presented in this book. We will present our approach to learning control programs by synthesizing functional programs in detail in part II. Other approaches were presented by (Shavlik, 1990), who applies inductive logic programming to control program learning, and by (Koza, 1992) in the context of genetic programming. Recently, learning recursive ("open loop") macros is also investigated in reinforcement learning (Kalmar and Szepesvari, 1999; Sun and Sessions, 1999).

While control rules guide search for a plan, control programs eliminate search completely. In our approach, we learn recursive functions for a domain of arbitrary complexity but with a fixed goal. For example, a program for constructing a tower of sorted blocks can be synthesized from a (universal) plan for a three block problem with goal $\{$on(A, B), on(B, C)$\}$. The synthe-

sized function then can generate correct and optimal action sequences for *tower* problems with an arbitrary number of blocks. Instead of searching for a plan needing probably exponential effort, the learned program is executed. Execution time corresponds to the complexity class of the program – for example linear time for a linear recursion. Furthermore, the learned control programs provide optimal action sequences.

While learning a control program for achieving a certain kind of top-level goals in a domain results in a highly efficient generation of an optimal action sequence, this might not be true if control knowledge is only learned for a subdomain – as for example, clearing a block as subproblem to building a tower of blocks. In this case, the problem of intelligent indexing and retrieval of control programs has to be dealt with – similar as in reuse of already generated plans in the context of a new planning problem (Veloso, 1994; Nebel and Koehler, 1995). Furthermore, program execution might lead to a state which is not on the optimal path of the global problem solution (Kalmar and Szepesvari, 1999).

Chapter 3

# CONSTRUCTING COMPLETE SETS OF OPTIMAL PLANS

The planning system DPlan is designed as a tool to support the first step of inductive program synthesis – generating finite programs for transforming input examples into the desired output. Because our work is in the context of program synthesis, planning is for small, deterministic domains and completeness and optimality are of more concern than efficiency considerations. In the remaining chapters of part I, we present DPlan as planning system. In part II we will describe, how recursive functions can be induced from plans constructed with DPlan and show how such recursive functions can be used as control programs for plan construction. In this chapter, we will introduce DPlan (sect. 1), introduce universal plans as sets of optimal plans (sect. 2), and give proofs for termination, soundness and completeness of DPlan (sect. 3).[1] An extension of DPlan to function application – allowing planning for infinite domains – is described in the next chapter (chap. 4).

## 1 INTRODUCTION TO DPLAN

DPlan[2] is a state-based, non-linear, total-order backward planner. DPlan is a universal planner (see sect. 4.4 in chap. 2): 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 plan, representing optimal action sequences for *all* states belonging to the planning problem. A short history of DPlan is given in appendix A1.

DPlan differs from standard universal planning in the following aspects:

[1] The chapter is based on the following previous publications: Schmid (1999), Schmid and Wysotzki (2000b, 2000a).

[2] Our algorithm is named DPlan in reference to the Dijkstra-algorithm (Cormen et al., 1990), because it is a single-source shortest-paths algorithm with the state(s) fulfilling the top-level goals as source.

- In contrast to universal planning for non-deterministic domains, we do not extract state-action rules from the universal plan, but use it as starting-point for program synthesis.

- A planning problem specifies a set of top-level goals and it restricts the number of objects of the domain, but *no* initial state is given. We are interested in optimal transformation sequences for all possible states from which the goal is reachable.

- A universal plan represents the optimal solutions for all states which can be transformed into a state fulfilling the top-level goal. That is, plan construction does not terminate if a given initial state is included in the plan, but only if expansion of the current leaf nodes does not result in new states (which are not already contained in the plan).

- A universal plan is represented as "minimal spanning DAG" (directed acyclic graph) (see sect. 2) with nodes as states and arcs as actions.[3] Instead of a memory-efficient representation using OBDDs, an explicit state-based representation is used as starting-point for program synthesis. Typically, plans involving only three or four objects are generated and a control program for dealing with $n$ objects is generalized by program synthesis.

- Backward operator application is not realized by goal regression over partial state descriptions (see sect. 3.4.1) but over complete state descriptions. The first step of plan construction expands the top-level goals to a set of goal states.

## 1.1 DPLAN PLANNING LANGUAGE

The current system is based on domain and problem specifications in PDDL syntax (see sect. 2.1 in chap. 2), allowing Strips (see def. 2.1) and operators with conditional effects. As usual, states are defined as sets of atoms (def. 2.2) and goals as sets of literals (def. 2.4). Operators are defined with preconditions, ADD- and DEL-lists (def. 2.5). Secondary preconditions can be introduced to model context-dependent effects (see sect. 2.1.2 in chap. 2).

Forward operator application for executing a plan by transforming an initial state into a goal state is defined as usual (see def. 2.6). Backward operator application is defined for complete state descriptions (see def. 2.7). An extension for conditional effects is given in definition 2.8. For universal planning, backward operator application must be extended to calculate the set of *all* predecessor states for a given state:

[3] As we will see for some examples below, for some planning domains, the universal plan is a specialized DAG – a minimal spanning tree or simply a sequence.

**Definition 3.1 (Pre-Image)** *With $Res^{-1}(o, s)$ we denote backward application of an instantiated operator $o$ to a state description $s$. We call $s' = Res^{-1}(o, s)$ a predecessor of $s$. Backward operator application can be extended in the following way:*

- *$Res_p^{-1}(\{o_1 \dots o_n\}, s) = \{s'_1, \dots, s'_m\}, m \leq n$ represents the "parallel" application of the set of all actions which satisfy the application condition for state $s$ resulting in a set of predecessor states.*
- *The pre-image of a set of states $S = \{s_1, \dots, s_l\}$ is defined as $\bigcup_{i=1}^{l} Res_p^{-1}(\{o_1 \dots o_n\}, s_i)$.*

To make sure that plan construction based on calculating pre-images is sound and complete, it must be shown that the pre-image of a set of states $S$ only contains consistent state descriptions and that the pre-image contains all state descriptions which can be transformed into a state in $S$ (see sect. 3).

A DPlan planning problem is defined as

**Definition 3.2 (DPlan Planning Problem)** *A planning problem $\mathcal{P}(\mathcal{O}, \mathcal{G}, \mathcal{D})$ consists of a set of operators $\mathcal{O}$, a set of top-level goals $\mathcal{G}$, and a domain restriction $\mathcal{D}$ which can be specified in two ways:*

- *as set of state descriptions,*
- *as a set of goal states.*

In standard planning and universal planning, an initial state is given as part of a planning problem. A planning problem is solved if an action sequence transforming the initial state into a state satisfying the top-level goals is found. In DPlan planning, a planning problem is solved if all states given in $\mathcal{D}$ are included in the universal plan, or if a set of goal states is expanded to all possible states from which a goal state is reachable.

The initial state has the following additional functions in backward plan construction (see sect. 3.4 in chap. 2): Plan construction terminates, if the initial state is included in the search tree. All partial state descriptions on the path from the initial state to the top-level goals can be completed by forward operator application. Consistency of state descriptions can be checked for the states included in the solution path and inconsistencies on other paths have no influence on the soundness of the plan. To make backward planning sound and complete, in general at least one complete state description – denoting a set of consistent relations between all objects of interest – is necessary. For DPlan, this must be either a (set of) complete goal state(s) or the set of all states to be included in the universal plan. If for a DPlan planning problem $\mathcal{D}$ is given as a set of goal states, consistency of a predecessor state can be checked by $Res^{-1}(o, s) = s' \rightarrow Res(o, s') = s$. If $\mathcal{D}$ is given as a set of (consistent) state

*Table 3.1.* Abstract DPlan Algorithm

**function** DPLAN(P) where *P* is a DPlan planning problem.

```
CurrentStates := ∅;
NextStates := GOALSTATES(P);
Plan := ∅;
while (NextStates ≠ CurrentStates) do
    OneStepPlan := ONESTEPPLAN(NextStates,P);
    Plan := Plan ∪ PRUNESTATES(OneStepPlan,NextStates);
    CurrentStates := NextStates;
    NextStates := NextStates ∪ PROJECTACTIONS(OneStepPlan);
return Plan.
```

description, plan construction is reduced to stepwise including states from $\mathcal{D}$ in the plan.

## 1.2 DPLAN ALGORITHM

The DPlan algorithm is a variant of the universal planning algorithm given in table 2.4. In table 3.1, the DPlan algorithm is described abstractly. Input is a DPlan planning problem $P(\mathcal{O}, \mathcal{G}, \mathcal{D})$ with a set of operators $\mathcal{O}$, a set of top-level goals $\mathcal{G}$ and a domain restriction $\mathcal{D}$ as specified in definition 3.2. Output is a universal plan, representing the union of all optimal plans for the restricted domain and a fixed goal, which we will describe in detail in section 2. In the following, the functions used for plan construction are described.

- GOALSTATES**(P)**:
  - If $\mathcal{D}$ is defined as set of goal states,
    GOALSTATES*(P)* := $\mathcal{D}$.
  - If $\mathcal{D}$ is defined as set of states $S$,
    GOALSTATES*(P)* := $\{s_G \mid \mathcal{G} \subseteq s_G \; and \; s_G \in S\}$.
- ONESTEPPLAN**(States, P)**:
  Calculates the pre-image of *States* as described in definition 3.1. Each state $s \in States$ is expanded to all states $s'$ with $Res(o, s') = s$. States $s'$ are calculated by backward operator application $Res^{-1}(o, s) = s'$ where all states for which $Res(o, s') = s$ does not hold are removed. If $\mathcal{D}$ is given as a set of states, additionally, all states not occurring in $\mathcal{D}$ are removed.
  *OneStepPlan* returns all pairs $\langle o, s' \rangle$:
  ONESTEPPLAN*(States, P)* = $\{\langle o, s' \rangle \mid Res(o, s) = s' \wedge s \in States\}$.

- PRUNESTATES**(Pairs, States)**:
  Eliminates all pairs $\langle o, s' \rangle$ from *Pairs* which have already been visited:
  PRUNESTATES*(Pairs, States)* := $\{\langle o, s' \rangle \in Pairs \; with \; s' \notin States\}$.

- PROJECTACTIONS**(Pairs)**:
  Returns the set of all states occurring in *Pairs*:
  PROJECTACTIONS*(Pairs)* := $\{s' \mid \langle o, s' \rangle \in Pairs\}$.

Plan construction corresponds to building a search-tree with breadth-first search with filtering of states which already occur on higher levels of the tree. Since it is possible that a *OneStepPlan* contains pairs $\langle o, s' \rangle$, $\langle o', s' \rangle$ – i. e., different actions resulting in the same predecessor state – the plan corresponds not to a tree but to a DAG (Christofides, 1975).

## 1.3 EFFICIENCY CONCERNS

Since planning is an NP-complete or even PSPACE complete problem (see sect. 3.3.1 in chap. 2), the worst-case performance of every possible algorithm is exponential. Nevertheless, current Graphplan-based and compilation approaches (see sect. 4.2.1 in chap. 2) show good performance for many benchmark problems. These algorithms are based on depth-first search and therefore in general cannot guarantee optimality – i. e., plans with a minimal length sequence of actions (Koehler et al., 1997). In contrast, universal planning is based on breadth-first search and can guarantee optimality (for deterministic domains). The main disadvantage of universal planning is that plan size grows exponentially for many domains (see sect. 3.3.1 in chap. 2). Encoding plans as OBDDs can often result in compact representations. But, the size of an OBDD is dependent on variable ordering (see sect. 4.4 in chap. 2) and calculating an optimal variable ordering is itself an NP-hard problem (Bryant, 1986).

Planning with DPlan is neither time nor memory efficient. 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[4], the effort of plan construction is dependent on the number of states. The number of states can grow exponentially as shown for example for the blocks-world domain in section 3.3.1 in chapter 2. Plans are represented as DAGs with state descriptions as nodes – that is, the size of the plan depends on the number of states, too.[5]

DPlan incorporates none of the state-of-the art techniques for efficient plan construction: we do use no pre-planning analysis (see sect. 5.1) and we do

[4] similar to dynamic programming in $A^*$, (Nilsson, 1980)

[5] For example, calculating an universal plan with the uncompiled Lisp implementation of DPlan, needs about a second for blocks-world with three objects, about 150 seconds for four objects, and already more than an hour for 5 objects.

| | |
|---|---|
| **Operator:** | **puttable(x)** |
| **PRE:** | {clear(x), on(x, y)} |
| **ADD:** | {ontable(x), clear(y)} |
| **DEL:** | {on(x, y)} |
| | |
| **Goal:** | {clear(C)} |
| **Dom-S:** | {{on(A, B), on(B, C), clear(A), ontable(C)}, |
| | {on(B, C), clear(A), clear(B), ontable(A), ontable(C)}, |
| | {clear(A), clear(B), clear(C), ontable(A), ontable(B), ontable(C)}} |

*Figure 3.1.* The *Clearblock* DPlan Problem

not encode plans as OBDDs. The reason is that we are mainly interested in an explicit representation of the structure of a domain with a fixed goal and a small number of domain objects. A universal plan constructed with DPlan represents the structural relations between optimal transformation sequences for all possible states. This information is necessary for inducing a recursive control program, generalizing over the number of domain objects as we will describe in chapter 8.

## 1.4 EXAMPLE PROBLEMS

In the following, we will give some example domains together with a DPlan problem, that is, a fixed goal and a fixed number of objects. For each DPlan problem, we present the universal plans constructed by DPlan. For better readability, we abstract somewhat from the LISP-based PDDL encoding which is input in the DPlan system (see appendix A3).

### 1.4.1 THE CLEARBLOCK PROBLEM

First we look at a very simple sub-domain of blocks-world (see fig. 3.1):. There is only one operator – *puttable* – which puts a block $x$ from another block on the table. We give the restricted domain *Dom* as a set of three states – a tower of $A$, $B$, $C$, a tower of $B$, $C$, and block $A$ lying on the table, and a state where all three blocks are lying on the table.

The planning goal is to clear block $C$. *Dom* includes one state fulfilling the goal and this state becomes the root of the plan. Plan construction results in ordering the three states given in *Dom* with respect to the length of the optimal transformation sequence (see fig. 3.2). For this simple problem, the plan is a simple sequence, i. e., the states are totally ordered with respect to the given goal and the given operator. Note that we present plans with states and actions in Lisp notation – because we use the original DPlan output (see appendix A2).

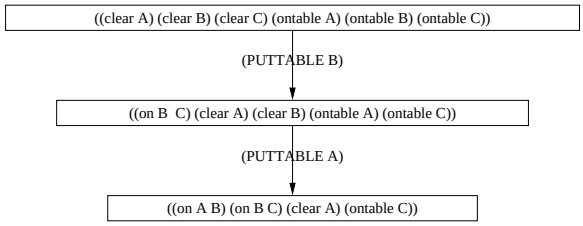
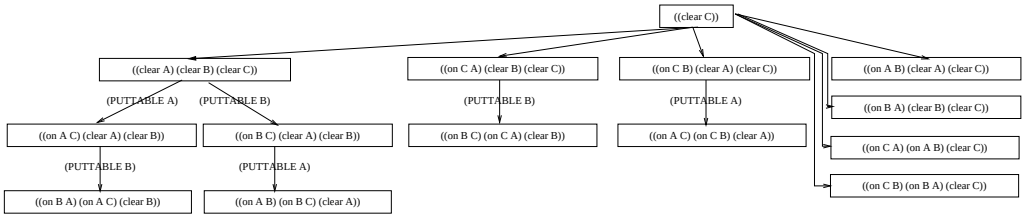



*Figure 3.2.* DPlan Plan for *Clearblock*




*(ontable omitted for better readability)*

*Figure 3.3.* *Clearblock* with a Set of Goal States

Alternatively, the set of all goal states satsifying *clear(C)* can be presented. Then, the resulting plan (see fig. 3.3) is a forest. When planning for multiple goal-states, we introduce the top-level goals as root.

### 1.4.2 THE ROCKET PROBLEM

An example for a simple transportation (logistics) domain is the *rocket* domain proposed by Veloso and Carbonell (1993) (see fig. 3.4). The planning goal is to transport three objects $O1$, $O2$, and $O3$ from a place $A$ to a destination $B$. The transport vehicle (*Rocket*) can only be moved in one direction ($A$ to $B$), for example from the earth to the moon. Therefore, it is important to load *all* objects before the rocket moves to its destination. The *rocket* domain was introduced as a demonstration for the incompleteness of linear planning (see sect. 3.4.2 in chap. 2).

The resulting universal plan is given in figure 3.5. For *rocket*, the plan corresponds to a DAG: loading and unloading of the three objects can be performed in an arbitrary sequence, but finally, each sequence results in all objects being inside the rocket or at the destination.

### 1.4.3 THE SORTING PROBLEM

A specification of a sorting problem is given in figure 3.6. The relational symbol *isc(p k)* represents that an element $k$ is at a position $p$ of a list (or more

**Operators:**

| | |
|---|---|
| | **load(?o, ?l)** |
| **PRE:** | {at(?o ?l), at(Rocket, ?l)} |
| **ADD:** | {inside(?o, Rocket)} |
| **DEL:** | {at(?o, ?l)} |
| | **move-rocket()** |
| **PRE:** | {at(Rocket, A)} |
| **ADD:** | {at(Rocket, B)} |
| **DEL:** | {at(Rocket, A)} |
| | **unload(?o, ?l)** |
| **PRE:** | {inside(?o, Rocket), at(Rocket, ?l)} |
| **ADD:** | {at(?o, ?l)} |
| **DEL:** | {inside(?o, Rocket)} |
| | |
| **Goal:** | {at(O1, B), at(O2, B), at(O3, B)} |
| **Dom-G:** | {{at(O1, B), at(O2, B), at(O3, B), at(Rocket, B)}} |

*Figure 3.4.* The DPlan *Rocket* Problem


($in = inside$, $R = rocket$)

*Figure 3.5.* Universal Plan for *Rocket*

| | |
|---|---|
| **Operator:** | **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)} |
| | |
| **Goal:** | {isc(p1, 1), isc(p2, 2), isc(p3, 3)} |
| **Dom-G:** | {{isc(p1, 1), isc(p2, 2), isc(p3, 3), gt(4, 3), gt(4, 2), gt(4, 1), gt(3, 2), gt(3, 1), gt(2, 1)} |

*Figure 3.6.* The DPlan *Sorting* Problem

exactly an array). The relational symbol *gt(x y)* represents that an element $x$ has a greater value than an element $y$, that is, the ordering on natural numbers is represented extensionally. Relational symbol $gt$ is static, i. e., its truth value is never changed by operator application. Because swapping of two elements is only restricted such that two elements are swapped if the first is greater than the second, the resulting plan corresponds to a set of traces of a selection sort program. We will discuss synthesis of selection sort in detail in chapter 8.

The universal plan for *sorting* is given in figure 3.7. For better readability, the lists itself instead of their logical description are given. In the chapter 4 we will describe how such lists can be manipulated directly by extending planning to function application.

### 1.4.4 THE HANOI PROBLEM

A specification of a Tower of Hanoi problem is given in figure 3.8. The Tower of Hanoi problem (chap. 5 Winston and Horn, 1989, e. g.,) is a well-researched puzzle. Given are $n$ discs (in our case three) with monotonical increasing size and three pegs. The goal is to have a tower of discs on a fixed peg (in our case $p_3$) where the discs are ordered by their size with the largest disc as base and the smallest as top element. Moving of a disc is restricted in the following way: Only a disc which has no other disc on its top can be moved. A disc can only moved onto a larger disc or an empty peg. 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). We will come back to this problem in chapter 8.

The plan for *hanoi* is given in figure 3.9. Again, states are represented abbreviated.

### 1.4.5 THE TOWER PROBLEM

The blocks-world domain with operators *put* and *puttable* was introduced in chapter 2 (see figs. 2.2, 2.3, 2.4, 2.5). We use a variant of the operator

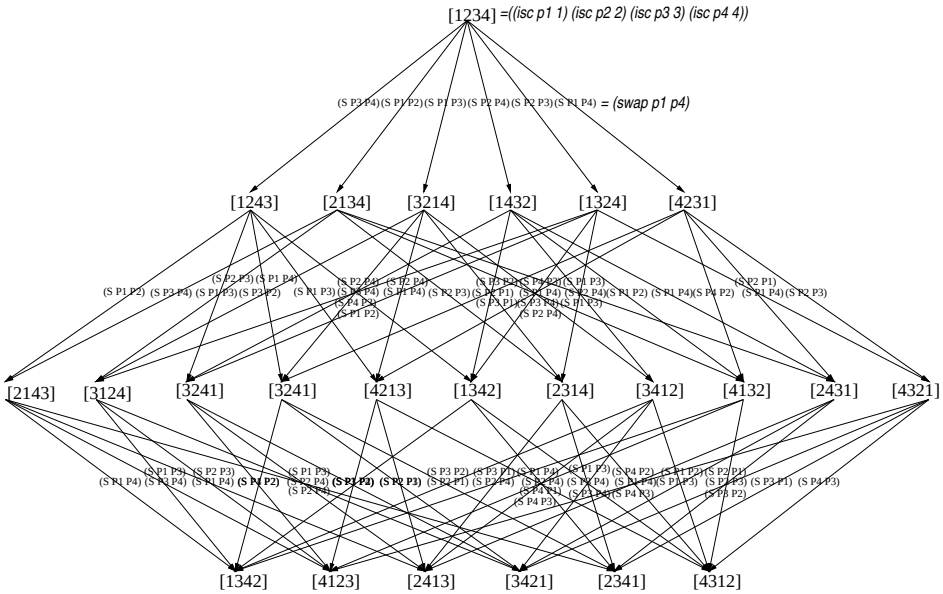



*Figure 3.7.* Universal Plan for *Sorting*

| | |
|---|---|
| **Operator:** | move(?d, ?from, ?to) |
| **PRE:** | {on(?d, ?from), clear(?d), clear(?to), smaller(?to, ?d)} |
| **ADD:** | {on(?d, ?to), clear(?from)} |
| **DEL:** | {on(?d, ?from), clear(?to)} |
| | |
| **Goal:** | {on($d_3$, $p_3$), on($d_2$, $d_3$), on($d_1$, $d_2$)} |
| **Dom-G:** | {on($d_3$, $p_3$), on($d_2$, $d_3$), on($d_1$, $d_2$), clear($d_1$), clear($p_1$), clear($p_2$)<br>smaller($p_1$, $d_1$), smaller($p_1$, $d_2$), smaller($p_1$, $d_3$),<br>smaller($p_2$, $d_1$), smaller($p_2$, $d_2$), smaller($p_2$, $d_3$),<br>smaller($p_3$, $d_1$), smaller($p_3$, $d_2$), smaller($p_3$, $d_3$),<br>smaller($d_1$, $d_2$), smaller($d_1$, $d_3$), smaller($d_2$, $d_3$)} |

*Figure 3.8.* The DPlan *Hanoi* Problem

definitions given in figure 2.2 where the both variants of *put* are specified as a single operator with conditioned effect. A plan for the goal {*on(A, B), on(B, C)*} is given in figure 3.10. We omitted from representing the *ontable* relations for better readability.

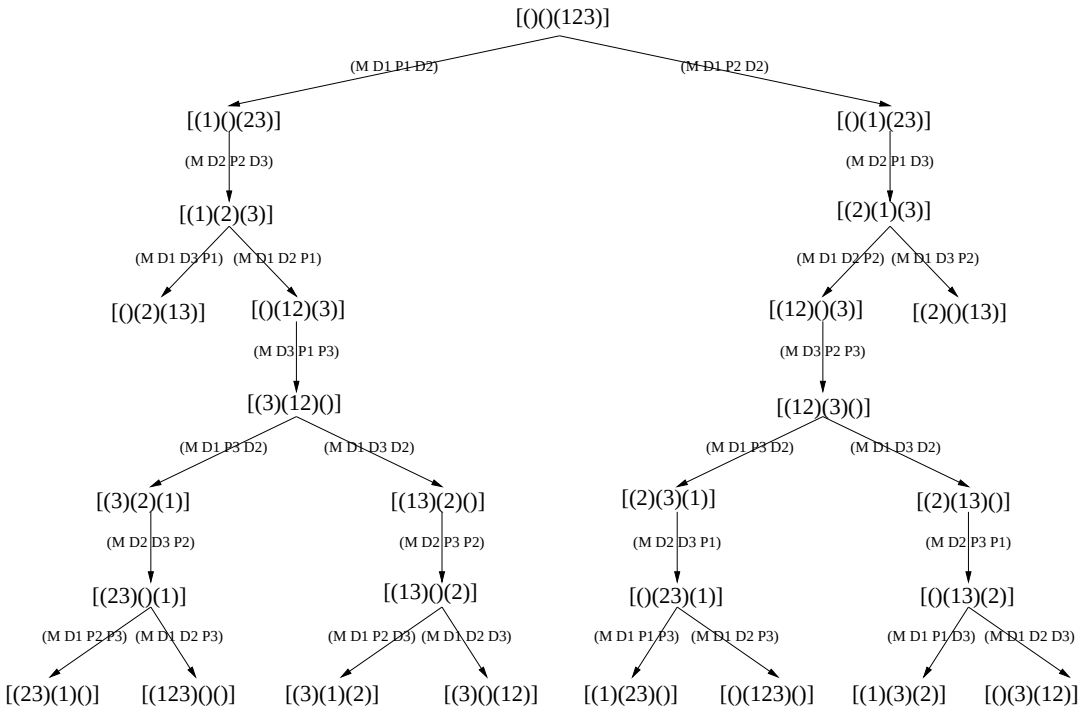



*Figure 3.9.* Universal Plan for *Hanoi*

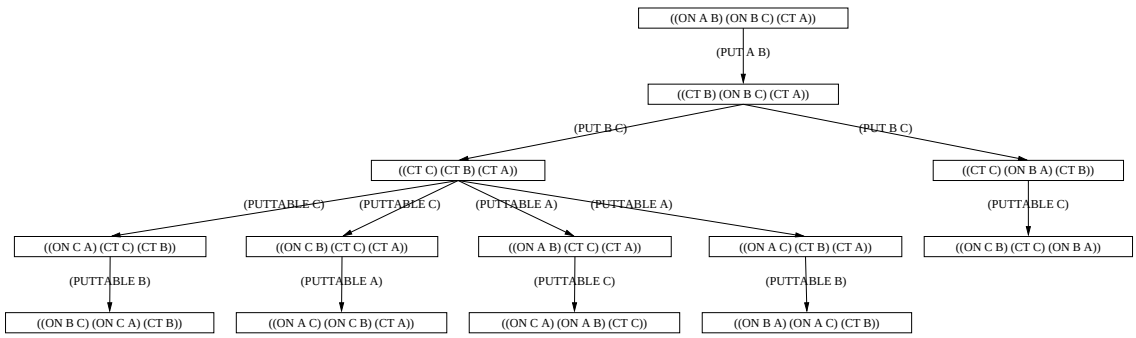



*Figure 3.10.* Universal Plan for *Tower*

## 2 OPTIMAL FULL UNIVERSAL PLANS

A DPlan plan, as constructed with the algorithm reported in table 3.1, is a set of action-state pairs $\langle o, s \rangle$. We define a plan as DAG in the following way:

**Definition 3.3 (Universal Plan)** *A set of action-state pairs $\mathcal{S} = \{\langle o, s\rangle\}$ constructed by DPlan corresponds to a directed acyclic graph (DAG).*

- *The set of states $S = \text{PROJECTACTIONS}(\mathcal{S})$ constitutes the set of nodes. Nodes $s$ with $\langle nil, s\rangle$, corresponding to goal states, are root nodes.*
- *Each pair of action-state pairs $\langle o, s\rangle$, $\langle o', s'\rangle$ with $o \neq nil$ constitutes an edge $r$, with $R$ as set of edges.*
- *An universal plan is defined as $\Pi^* = (S, R)$.*[6]

A universal plan $\Pi^*$ represents an order over the state it contains with respect to the minimal length of the number of actions needed to transform a state into a goal state:

**Definition 3.4 (Universal Plan as Order over States)** *We define a path $\Pi \in \Pi^*$ as a sequence $(\langle nil, s_0\rangle, \langle o_1, s_1\rangle, \ldots, \langle o_n, s_n\rangle)$.*

- *The length of a path $\Pi$ corresponds to the number of pairs it contains:* length($\Pi$) = $\mid \Pi \mid$ -1.
- *The number of steps to transform the leaf node $s_n$ of a path $\Pi$ into a goal state $s_0$ is $g(s_n) = length(\Pi)$.*
- *A plan $\Pi^*$ defines an order over the set of states $S$: $s_i < s_j$ iff $g(s_i) < g(s_j)$. Because $\Pi^*$ is constructed in accordance to the DPlan algorithm, a pair $\langle o, s\rangle$ is only introduced if $s$ is not already contained at a higher level of the plan. Although there might be pairs $\langle o, s\rangle$, $\langle o', s\rangle$, a state $s$ can only occur at a fixed level of $\Pi^*$. That is $g(s)$ is unique for all $s \in S$ and states are totally ordered with respect to $g(s)$.*

In contrast to standard universal planning, DPlan constructs a plan which contains all states which can be transformed into a state satisfying the given top-level goals:

**Definition 3.5 (Full Universal Plan)** *A universal plan $\Pi^*$ is called* full *with respect to given top-level goals $\mathcal{G}$ and a restricted number of domain objects $\mathcal{C}$, if all states $s_n$ for which a transformation sequence $Res(o_n, s_n)$, $Res(o_{n-1}, s_{n-1})$, ..., $Res(o_1, s_1)$ with $\mathcal{G} \subseteq Res(o_1, s_1)$ exist are contained in $\Pi^*$.*

For example, the universal plan for *rocket* (see fig. 3.5) defined for three objects and a single rocket, $\mathcal{C} = \{O1, O2, O3, Rocket\}$ contains transformation sequences for all legal states which can be defined over $\mathcal{C}$.

[6] In contrast to the standard definition of graphs (Christofides, 1975), we do not discriminate between nodes/edges and labels of nodes/edges.

The notion of a full universal plan presupposes that planning with DPlan must be complete. Additionally, we require that the universal plan does not contain states which cannot be transformed into a goal state, that is, DPlan must be sound. Completeness and soundness of DPlan is discussed in section 3. For proving optimality of DPlan we also assume completeness.

**Lemma 3.1 (Universal Plans are Optimal)**
*Presupposing that DPlan is complete – i. e., for each state $s'$ with $Res(o, s') = s$ holds that $s'$ is in the pre-image of a set of states $S$ with $s \in S$ – each path $\Pi$ of the universal plan $\Pi^*$ which leads to $s_n$ represents the minimal number of actions which transform $s_n$ into a goal state $s_0$.*

**Proof 3.1 (Universal Plans are Optimal)**
*We give an informal proof by induction over the levels of $\Pi^*$:*

- *Plan construction starts with a set of goal states $S_G = \{s \mid \mathcal{G} \subseteq s\}$. These states are introduced in $\Pi^*$ as pairs $\langle nil, s\rangle$. Paths $\Pi = (\langle nil, s_0\rangle)$ have length 0, that is* g(s) = 0 *for all states in $S_G$.*

- *Given a universal plan $\Pi^*$ consisting of $n$ levels. Assuming completeness of DPlan, $\Pi^*$ contains* all *states $s$ which can be transformed into a goal state with maximally $n-1$ actions. Let $S$ be the set of nodes at level $n$ in $\Pi^*$. For all $s \in S$ holds $g(s) \leq n-1$.*
  *Let $S'$ be the pre-image of $S$ calculated by DPlan. If $s' \in S'$ is already in $\Pi^*$ then it gets removed from $S'$ by definition of* PRUNESTATES. *If $s' \in S'$ is not in $\Pi^*$ then there exists no action sequence transforming $s'$ in a goal state with $g(s') \leq n-1$. The shortest action sequence has length $g(s') = n$.*

The length of minimal paths in a graph constitutes a metric. That is, for all paths holds that if the transformation sequence from start to goal node of the path is optimal then the transformation sequences for all intermediate nodes on the path are optimal, too.

As a consequence of the optimality of DPlan, each path in $\Pi^*$ from some state to a goal state represents an optimal solution for this state. In general, there might be multiple optimal paths from a state to the goal. For example, for *rocket* (see sect. 1.4.2) there are 3! different possible sequences for unloading and loading three objects. For the leaf node in figure 3.5 there are 36 different optimal solution paths contained in the universal plan.

A DPlan plan can be reduced to a minimal spanning tree (Christofides, 1975) by deleting edges such that each node which is not the root node has exactly one predecessor. The minimal spanning tree contains exactly one optimal solution path for each state. A minimal spanning tree for *rocket* is given in figure 3.11.

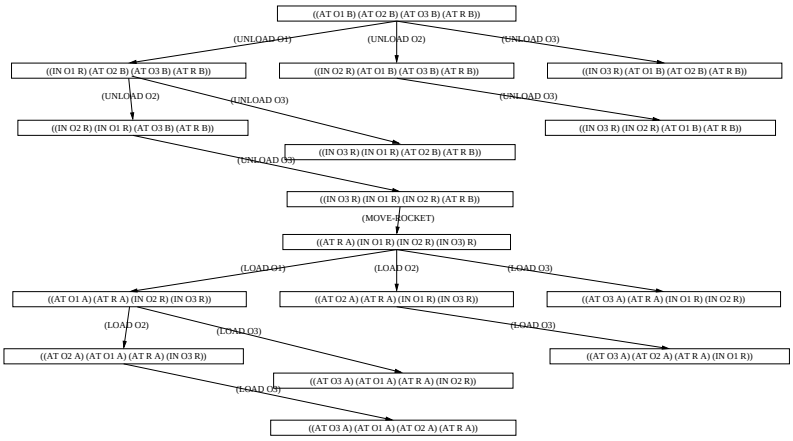



*Figure 3.11.* Minimal Spanning Tree for *Rocket*

# 3 TERMINATION, SOUNDNESS, COMPLETENESS

## 3.1 TERMINATION OF DPLAN

For finite domains, termination of the DPlan algorithm given in table 3.1 is guaranteed:

**Lemma 3.2 (Termination of DPlan)**
*For finite domains, a situation* (NextStates = CurrentStates) *will be reached because the growth of the number of states in* Plan *is monotonical.*

**Proof 3.2 (Termination of DPlan)**
*We denote a plan after the $t$-th iteration of pre-image calculation with $Plan_t$. From the definition of the DPlan algorithm follows that* NextStates = PROJECTACTIONS(Plan).

- *If calculating a pre-image only returns state descriptions which are already contained in plan $Plan_t$ then* (NextStates = CurrentStates) *holds and DPlan terminates after $t$ iterations.*

- *If calculating a pre-image returns at least one state description which is not already contained in the plan then this state description(s) are included in the plan and $\mid Plan_{t+1} \mid > \mid Plan_t \mid$ .*

- *Because planning domains are assumed to be finite, there is only a fixed number of different state descriptions and a fixed point $\mid Plan_{t+1} \mid = \mid Plan_t \mid$ , i. e., a situation* (NextStates = CurrentStates) *will be reached after a finite number of steps $t$.*

## 3.2 OPERATOR RESTRICTIONS

Soundness and completeness of plan construction is determined by the soundness and completeness of operator application. 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 definitions 2.6 and 2.7:

$$Res(o,s) = s \setminus DEL \cup ADD \; if \; PRE \subseteq s$$

$$Res^{-1}(o,s) = s \setminus ADD \cup (DEL \cup PRE) \; if \; ADD \subseteq s.$$

For forward application, $\setminus DEL$ and $\cup ADD$ are only commutative for $ADD \cap DEL = \emptyset$[7]. When literals from the DEL-list are deleted before adding literals from the ADD-list, 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 must commute:

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

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

**Soundness:**

$$\begin{array}{lcl} Res(o, Res^{-1}(o,s)) & \stackrel{?}{=} & s \\ Res(o, Res^{-1}(o,s)) & = & \\ & = & Res^{-1}(o,s) \setminus DEL \cup ADD \\ & & with \; PRE \subseteq Res^{-1}(o,s) \\ & = & \{s \setminus ADD \cup \{DEL \cup PRE\}\} \setminus DEL \cup ADD \\ & & with \; PRE \subseteq \{s \setminus ADD \cup \{DEL \cup PRE\}\} \\ & & and \; ADD \subseteq s \end{array}$$

[7] In PDDL $ADD \cap DEL = \emptyset$ must be always true because the effects are given in a single list with DEL-effects as negated literals. Therefore, an expression `(and p (not p))` would represent a contradiction and cannot be not allowed as specification of an operator effect.

$$
\begin{aligned}
&= \{s \cup \{DEL \cup PRE\}\} \setminus DEL \\
&\quad with\ PRE \subseteq \{s \setminus ADD \cup \{DEL \cup PRE\}\} \\
&= \{s \cup DEL\} \setminus DEL\ if\ PRE \subseteq s \cup DEL \\
&= s\ if\ s \cap DEL = \emptyset.
\end{aligned}
$$

The restriction $PRE \subseteq s \cup DEL$ means for forward operator application, transforming from $s'$ into $s$, that $PRE$ holds after operator application if it is not explicitly deleted. In backward application $PRE$ is introduced as list of sub-goals and constraints which have to hold in $s'$. Therefore, this restriction is unproblematic. The restriction $s \cap DEL = \emptyset$ means for forward application that $s$ contains no literal from the DEL-list. For backward-planning all literals from the DEL-list are added and $s$ has contained none of these literals. This restriction is in accordance with the definition of legal operators. Thus, soundness of backward application of an Strips operator is given. In general, soundness of backward-planning can be guaranteed by introducing an consistency check for each constructed predecessor: For a newly constructed state $s'$, $s = Res(o, s')$ must hold. If forward operator application does not result in $s$, the constructed predecessor is considered as not admissible and not introduced in the plan.

**Completeness:**

$$
\begin{aligned}
Res^{-1}(o, Res(o, s')) &\stackrel{?}{=} s' \\
Res^{-1}(o, Res(o, s')) &= \\
&= Res(o, s') \setminus ADD \cup \{DEL \cup PRE\} \\
&\quad with\ ADD \subseteq Res(o, s') \\
&= (s' \setminus DEL \cup ADD) \setminus ADD \cup \{DEL \cup PRE\} \\
&\quad with\ ADD \subseteq (s' \setminus DEL \cup ADD) \\
&\quad and\ with\ PRE \subseteq s' \\
&= s' \setminus DEL \cup \{DEL \cup PRE\} \\
&\quad with\ PRE \subseteq s'\ if\ s' \cap ADD = \emptyset \\
&= s' \cup PRE\ with\ PRE \subseteq s'\ if\ DEL \subseteq s' \\
&= s'.
\end{aligned}
$$

The restriction $s' \cap ADD = \emptyset$ means that only such states can be constructed which do not already contain a literal added by an applicable operator. The restriction $DEL \subseteq s'$ means that only such states can be constructed 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. In general, incompleteness can be overcome – with a

loss of efficiency – by constructing predecessors in the following way:

$$Res^{-1}(o, s) = s \setminus ADD^* \cup \{DEL^* \cup PRE\}$$

for all combinations of subsets $ADD^*$ of $ADD$ and $DEL^*$ of $DEL$ if $ADD \subseteq s$. Thus, all possible states containing subsets of $DEL$ and $ADD$ with the special case of inserting all literals from $DEL$ and deleting all literals from $ADD$, would be constructed. Of course, most of these states might not be sound!

For more general operator definitions, as allowed by PDDL, further restrictions arise. For example, it is necessary, that secondary preconditions are mutually exclusive.

Another definition of sound and complete backward operator application is given by (Haslum and Geffner, 2000): For $s \cap ADD \neq \emptyset$ and $s \cap DEL = \emptyset$: $Res^{-1}(o, s) = s \setminus ADD \cup PRE$. For $DEL \subseteq PRE$, this definition is equivalent with our definition.

## 3.3 SOUNDNESS AND COMPLETENESS OF DPLAN

**Definition 3.6 (Soundness of DPlan)** *A universal plan $\Pi^*$ is sound, if each path $\Pi = \{\langle nil, s_0\rangle, \langle o_1, s_1\rangle, \ldots \langle o_n, s_n\rangle\}$ defines a solution. That is, for an initial state corresponding to $s_n$, the sequence of actions $o_n \circ \ldots \circ o_1$ transforms $s_n$ into a goal state.*

Soundness of DPlan follows from the soundness of backward operator application as defined in section 3.2.

**Definition 3.7 (Completeness of DPlan)** *A universal plan $\Pi^*$ is complete, if the plan contains all states $s$ for which an action sequence, transforming $s$ into a goal state, exists.*

Completeness of DPlan follows from the completeness of backward operator application as defined in section 3.2.

If DPlan works on a domain restriction $\mathcal{D}$ given as set of states, only such states are included in the plan which also occur in $\mathcal{D}$. From soundness and completeness of DPlan also follows that for a given set of states $\mathcal{D}$, DPlan terminates with *dom* = PROJECTACTIONS*(Plan)* if the state space underlying $\mathcal{D}$ is (strongly) connected.[8] If DPlan terminates with PROJECTACTIONS*(Plan)* $\subseteq \mathcal{D}$ then $\mathcal{D} \setminus$ PROJECTACTIONS*(Plan)* only contains states from which no goal-state in PROJECTACTIONS*(Plan)* is reachable. One reason for non-reachability can be that the state space is not connected. That is, it contains

[8] For definitions of connectedness of graphs, see for example (Christofides, 1975).

states on which no operator is applicable. For example, in the restricted blocks-world domain where only a *puttable* but no *put* operator is given (see sect. 1.4.1), no operator is applicable in a state where all blocks are lying on the table. As a consequence, a goal *on(x, y)* is not reachable from such a state. A second reason for non-reachability can be that the state space is weakly connected (i. e., is a directed graph) such that it contains states $s, s'$ where $Res(o, s) = s'$ is defined but not $Res(o', s') = s$. For example, in the *rocket* domain (see sect. 1.4.2), it is not possible to reach a state where the rocket is on the earth from a state where the rocket is on the moon. As a consequence, a goal *at(Rocket, Earth)* is nor reachable from any state where the rocket is on the moon.

Chapter 4

# INTEGRATING FUNCTION APPLICATION IN STATE-BASED PLANNING

Standard planning is defined for logical formulas over variables and constants. Allowing general terms – that is, application of arbitrary symbol-manipulating and numerical functions – enlarges the scope of problems for which plans can be constructed. In the context of using planning as starting point for the synthesis of functional programs, we are interested in applying planning to standard programming problems, such as list sorting. In the following, we will first give a motivation for extending planning to function application (sect. 1), then an extension of the Strips language presented in section 1 is introduced (sect. 2) together with some extensions (sect. 3), and finally examples for planning with function application are given (sect. 4).[1]

## 1 MOTIVATION

The development of efficient planning algorithms in the nineties (see sect. 4.2 in chap. 2) – such as Graphplan, SAT planning, and heuristic planning (Blum and Furst, 1997; Kautz and Selman, 1996; Bonet and Geffner, 1999) – has made it possible to apply planning to more demanding real-world domains. Examples are the logistics or the elevator domain (Bacchus et al., 2000). Many realistic domains involve manipulation of numerical objects. For example: when planning (optimal) routes for delivering objects as in the logistics domain, it might be necessary to take into account time and other resource constraints as fuel; plan construction for landing a space-vehicle involves calculations for the correct adjustments of thrusts (Pednault, 1987). One obvious way to deal with numerical objects is to assign and update their values by means of function applications.

[1] The chapter is based on the paper Schmid, Müller, and Wysotzki (2000).

There is some initial work on extending planning formalisms to deal with resource constraints (Koehler, 1998; Laborie and Ghallab, 1995). Here, function application is restricted to manipulation of specially marked resource variables in the operator effect. For example, the amount of fuel might be decreased in relation to the distance an airplane flies: *$gas -= distance(?x ?y)/3* (Koehler, 1998). A more general approach for including function applications into planning was proposed by (Pednault, 1987) within the action description language (ADL). In addition to ADD/DEL operator effects, ADL allows variable updates by assigning new values which can be calculated by arbitrary functions. For example, the *put(?x, ?y)* operator in a blocks-world domain might be extended to updating the state variable *?LastBlockMoved* to *?LastBlockMoved := ?x*; the amount of water in a jug $?j_2$ might be changed by a *pour($?j_1$, $?j_2$)* action into $?j_2$ := *min*$[?c_2, ?j_1 + ?j_2]$ where $?j_1$, $?j_2$ are the current quantities of water in the jugs and $?c_2$ is the capacity of jug $?j_2$ (Pednault, 1994).

Introducing functions into planning not only makes it possible to deal with numerical values in a more general way than allowed for by a purely relational language but has several additional advantages (see also Geffner, 2000) which we will illustrate with the Tower of Hanoi domain (see fig. 4.1)[2]: Allowing not only numerical but also symbol-manipulating functions makes it possible to model operators in a more compact and sometimes also more natural way. For example, representing which disks are lying on which peg in the Tower of Hanoi domain could be realized by a predicate *on(?disks, ?peg)* where $?disks$ represents the ordered sequence of disks on peg $?peg$ with list $[\infty]$ representing an empty peg. Instead of modeling a state change by adding and deleting literals from the current state, arguments of the predicates can be changed by applying standard list-manipulation functions (e. g., built-in Lisp functions like `car(l)` for returning the first element of a list, `cdr(l)` for returning a list without its first element, or `cons(x, l)` for inserting a symbol $x$ in front of a list $l$).

A further advantage is that objects can be referred to in an indirect way. In the *hanoi* example, `car(l)` refers to the object which is currently the uppermost disk on a peg. There is no need for any additional fluent predicate besides *on(?disks, ?peg)*. The *clear(?disk)* predicate given in the standard definition becomes superfluous. Geffner (2000) points out that indirect reference reduces substantially the number of possible ground atoms and in consequence the number of possible actions, thus plan construction becomes more efficient. Indirect object reference additionally allows for modeling infinite domains while the state representations remain small and compact. For example, `car(l)`

[2]Note that we use prefix notation $p(x_1, \ldots, x_n)$ for relations and functions throughout the paper.

**(a) Standard representation**

| | |
|---|---|
| **Operator:** | move(?d, ?from, ?to) |
| **PRE:** | {on(?d, ?from), clear(?d), clear(?to), smaller(?d, ?to)} |
| **ADD:** | {on(?d, ?to), clear(?from)} |
| **DEL:** | {on(?d, ?from), clear(?to)} |
| | |
| **Goal:** | {on($d_3$, $p_3$), on($d_2$, $d_3$), on($d_1$, $d_2$)} |
| **Initial State:** | {on($d_3$, $p_1$), on($d_2$, $d_3$), on($d_1$, $d_2$), clear($d_1$), clear($p_2$), clear($p_3$)<br>smaller($d_1$, $p_1$), smaller($d_1$, $p_2$), smaller($d_1$, $p_3$),<br>smaller($d_2$, $p_1$), smaller($d_2$, $p_2$), smaller($d_2$, $p_3$),<br>smaller($d_3$, $p_1$), smaller($d_3$, $p_2$), smaller($d_3$, $p_3$),<br>smaller($d_1$, $d_2$), smaller($d_1$, $d_3$), smaller($d_2$, $d_3$)} |

**(b) Functional representation**

| | |
|---|---|
| **Operator:** | move($?p_i$, $?p_j$) |
| **PRE:** | {on($?l_i$, $?p_i$), on($?l_j$, $?p_j$), (car($?l_i$) $\neq \infty$), car($?l_i$) $<$ car($?l_j$)} |
| **UPDATE:** | *change* $?l_j$ *in* on($?l_j$, $?p_j$) *to* cons(car($?l_i$),$?l_j$)<br>*change* $?l_i$ *in* on($?l_i$, $?p_i$) *to* cdr($?l_i$) |
| | |
| **Goal:** | {on([$\infty$], $p_1$), on([$\infty$], $p_2$), on([1 2 3 $\infty$], $p_3$)} ; $\infty$ represents a dummy |
| **Initial State:** | {on([1 2 3 $\infty$], $p_1$), on([$\infty$], $p_2$), on([$\infty$], $p_3$)} ; bottom disk |

*Figure 4.1.* *Tower of Hanoi* (a) Without and (b) With Function Application

gives us the top disk of a peg regardless of how many disks are involved in the planning problem.

Finally, introducing functions in modeling planning domains often makes it possible to get rid of static predicates. For example, the *smaller(?x, ?y)* predicate in the *hanoi* domain can be eliminated. By representing disks as numbers, the built-in predicate "$<$" can be used to check whether the application constraint for the *hanoi move* operator is satisfied in the current state. Allowing arbitrary boolean operators to express preconditions generalizes *matching* of literals to contraint satisfaction.

The need to extend standard relational specification languages to more expressive languages, resulting in planning algorithms applicable to a wider range of domains, is recognized in the planning community. The PDDL planning language for instance (McDermott, 1998b), the current standard for planners, incorporates all features which extend ADL over classical Strips. But while there is a clear operational semantics for conditional effects and quantification (Koehler et al., 1997), function application is dealt with in a largely ad-hoc manner (see also remark in Geffner, 2000).

The only published approach allowing functions in domain and problem specifications is Geffner (2000). He proposed Functional Strips, extending standard Strips to support first-order-function symbols. He gives a denotational

| | |
|---|---|
| **Domains:** | Peg: $p_1$, $p_2$, $p_3$ ; the pegs<br>Disk: $d_1$, $d_2$, $d_3$ ; the disks<br>Disk*: Disk, $d_0$ ; the disks and a dummy bottom disk 0 |
| **Fluents:** | top: Peg $\rightarrow$ Disk* ; denotes top disk in peg<br>loc: Disk $\rightarrow$ Disk*; denotes disk below given disk<br>size: Disk* $\rightarrow$ Integer ; represents disk size |
| **Action:** | move($?p_i$, $?p_j$ : Peg) ; moves between pegs |
| **Prec:** | top($?p_i$) $\neq d_0$, size(top($?p_i$)) $<$ size(top($?p_j$)) |
| **Post:** | top($?p_i$) := loc(top($?p_i$)); loc(top($?p_i$)) := top($?p_j$); top($?p_j$) := top($?p_i$) |
| **Init:** | loc($d_1$) = $d_0$, loc($d_2$) = $d_1$, loc($d_3$) = $d_2$, loc($d_4$) = $d_3$,<br>top($p_1$) = $d_4$, top($p_2$) = $d_0$, top($p_3$) = $d_0$,<br>size($d_0$) = 4, size($d_1$) = 3, size($d_2$) = 2, size($d_3$) = 1, size($d_4$) = 0 |
| **Goal:** | loc($d_1$) = $d_0$, loc($d_2$) = $d_1$, loc($d_3$) = $d_2$, loc($d_4$) = $d_3$, top($p_3$) = $d_4$ |

*Figure 4.2.* *Tower of Hanoi* in Functional Strips (Geffner, 1999)

state-based semantics for actions and an operational semantics, describing action effects as updates of state representations. In contrast to the classical relational languages, states are not represented as sets of (ground) literals but as *state variables* (cf. situation calculus). For example, the initial positon of disk $d_1$ is represented as *loc($d_1$)* = $d_0$ (see fig. 4.2).

While in Functional Strips relations are completely replaced by functional expressions, our approach allows a combination of relational and functional expressions. We extended the Strips planning language to a subset of first order logic, where relational symbols are defined over *terms* where terms can be variables, constant symbols, *or* functions with arbitrary terms as arguments. Our state representations are still sets of literals, but we introduce a second class of relational symbols (called *constraints*) which are defined over arbitrary functional expressions, as shown in figure 4.1. Standard representations containing only literals defined over constant symbols are included as special case, and we can combine ADD/DEL effects and updates in operator definitions.

## 2 EXTENDING STRIPS TO FUNCTION APPLICATIONS

In the following we introduce an extension of the Strips language (see def. 2.1) from propositional representations to a larger subset of first order logic, allowing general terms as arguments of relational symbols. Operators are defined over these more general formulas, resulting in a more complex state transition function.

**Definition 4.1 (FPlan Language)** *The language $\mathcal{L}(\mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R}_\mathcal{P} \cup \mathcal{R}_\mathcal{C})$ is defined over sets of variables $\mathcal{X}$, constant symbols $\mathcal{C}$,* **function symbols** *$\mathcal{F}$, and of relational symbols $\mathcal{R} = \mathcal{R}_\mathcal{P} \cup \mathcal{R}_\mathcal{C}$ in the following way:*

- *Variables $x \in \mathcal{X}$ are terms.*
- *Constant symbols $c \in \mathcal{C}$ are terms.*
- **If $f \in \mathcal{F}$ is a function symbol with arity $\alpha(f) = i$ and $t_1, t_2, \ldots\ t_i$ are terms, then $f(t_1, t_2, \ldots, t_i)$ is a term.**
- *If $p \in \mathcal{R}_\mathcal{P}$ is a relational symbol with arity $\alpha(p) = j$ and $t_1, t_2, \ldots\ t_j$ are terms then $p(t_1, t_2, \ldots, t_j)$ is a formula.*
  **If $r \in \mathcal{R}_\mathcal{C}$ is a relational symbol with arity $\alpha(r) = j$ and $t_1, t_2, \ldots\ t_j$ are terms then $r(t_1, t_2, \ldots, t_j)$ is a formula.** *We call formulas with relational symbols from $\mathcal{R}_\mathcal{C}$* constraints.
  *For short, we write $p(\vec{t})$.*
- *If $p_1(\vec{t_1}), p_2(\vec{t_2}), \ldots, p_k(\vec{t_k})$ with $p_1, p_2, \ldots, p_k \in \mathcal{R}$, then $\{p_1(\vec{t_1}), p_2(\vec{t_2}), \ldots, p_k(\vec{t_k})\}$ is a formula, representing the conjunction of literals.*
- *There are no other FPlan formulas.*

Remarks:

- *Formulas consisting of a single relational symbol are called (positive) literals.*
  *Terms over $\mathcal{C} \cup \mathcal{F}$, i. e., terms without variables, are called ground terms.*
  *Formulas over ground terms are called ground formulas.*
- *With $\mathcal{X}(F)$ we denote the variables occurring in formula $F$.*

An example of a state representation, goal definition, and operator definition using the FPlan language is given in figure 4.1.b for the *hanoi* domain. FPlan is instantiated in the following way: $\mathcal{X} = \{?p_i, ?p_j, ?l_i, ?l_j\}$, $\mathcal{C} = \{p_1, p_2, p_3, 1, 2, 3, \infty\}$, $\mathcal{F} = \{[\ ]^i, car^1, cdr^1, cons^2\}$, $\mathcal{R}_\mathcal{P} = \{on^2\}$, $\mathcal{R}_\mathcal{C} = \{\neq^2, <^2\}$.

A state in the *hanoi* domain, such as {*on([2 3 $\infty$], $p_1$), on([$\infty$], $p_2$), on([1 $\infty$], $p_3$)*}, is given as a set of atoms where the arguments can be arbitrary ground terms. We can use constructor functions to define complex data structures. For example, *[2 3 $\infty$]* is a list of constant symbols, where *[ ]* is a list constructor function defined over an arbitrary number of elements. The atom *on([2 3 $\infty$], $p_1$)* corresponds to the evaluated expression *on(cdr([1 2 3 $\infty$]), $p_1$)*, where *cdr(?l)* returns a list without its first element. State transformation by updating is defined by such evaluations.

Relational symbols are divided in to two categories – symbols in $\mathcal{R}_\mathcal{P}$ and symbols in $\mathcal{R}_\mathcal{C}$. Symbols in $\mathcal{R}_\mathcal{P}$ denote relations which characterize a problem

state. For example, *on([1 2 3 ∞], $p_1$)* with *on* $\in R_P$ is true in a state, where disks 1, 2, and 3 are lying on peg $p_1$. Symbols in $\mathcal{R}_\mathcal{C}$ denote additional characteristics of a state, which can be inferred from a formula over $\mathcal{R}_\mathcal{P}$. For example, *car($L_1$)* $\neq \infty$ with $\neq \in \mathcal{R}_\mathcal{C}$ is true for $L_1$ = *[1 2 3 ∞]*. Relations over $\mathcal{R}_\mathcal{C}$ often are useful for representing constraints, especially constraints which must hold for all states, that is static predicates.

Preconditions are defined as arbitrary formulas over $\mathcal{R}_\mathcal{P} \cup \mathcal{R}_\mathcal{C}$. Standard preconditions, such as *p = on($?l_1$, $?x$)*, can be transformed into constraints: If *match(p,s)* results in $\sigma_1 = \{l_1 \leftarrow [1\ 2\ 3\ \infty], ?x \leftarrow p_1\}$, then the constraints $eq(?x, p_1)$ and $eq(?l_1, [1\ 2\ 3\ \infty])$, with *eq* as equality-test for symbols in $\mathcal{R}_\mathcal{C}$, must hold.

We allow constraints to be defined over free variables, i. e., variables that cannot be instantiated by matching an operator precondition with a current state. For example, variables $?i$ and $?j$ in the constraint $nth(?i, ?l) < nth(?j, ?l))$ might be free. To restrict the possible instantiations of such variables, they must be declared together with a range in the problem specification (see sect. 4.5 for an example).

**Definition 4.2 (Free Variables in $\mathcal{R}_\mathcal{C}$)** *For a formula $F \in \mathcal{L}(\mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R}_\mathcal{P} \cup \mathcal{R}_\mathcal{C})$, variables occurring only in literals over $\mathcal{R}_\mathcal{C}$ are called free. The set of such variables is denoted as $\mathcal{X}_\mathcal{C} \subseteq \mathcal{X}$ and with $\mathcal{X}_\mathcal{C}(F)$ we refer to free variables in $F$.*
*For all variables $x$ in $\mathcal{X}_\mathcal{C}$ instantiations must be restricted by a* range $R(x)$. *For variables belonging to an ordered data type (e. g., natural numbers), a range is declared as $R(x) = [min, max]$ with* min *giving the smallest and* max *the largest value $x$ is allowed to assume. Alternatively, for categorial data types, a range is defined by enumerating all possible values the variable can assume: $R(x) = [v_1, \ldots, v_n]$.*

**Definition 4.3 (State Representation)** *A problem state $s$ is a conjunction of atoms over relational symbols in $\mathcal{R}_\mathcal{P}$. That is, $s \in \mathcal{L}(\mathcal{C}, \mathcal{F}, \mathcal{R}_\mathcal{P})$.*

We presuppose that terms are always *evaluated* if they are grounded. Evaluation of terms is defined in the usual way (Field and Harrison, 1988; Ehrig and Mahr, 1985):

**Definition 4.4 (Evaluation of Ground Terms)** *A term t over $\mathcal{L}(\mathcal{C}, \mathcal{F})$ is evaluated in the following way:*

- *eval(c) = c, for $c \in \mathcal{C}$*
- *eval($f(t_1, \ldots, t_n)$) = apply(f(eval($t_1$), ..., eval($t_n$))), for $f \in \mathcal{F}$ and $t_1, \ldots, t_n \in \mathcal{C} \cup \mathcal{F}$.*

*Function application returns a unique value for* $f(c_1, \ldots, c_n)$.
*For constructor functions (such as the list constructor [ ]) we define apply(*$f_c(c_1, \ldots, c_n)$*) =* $f_c(c_1, \ldots, c_n)$.

For example, *eval(cdr([1 2 3* $\infty$*]))* returns *[2 3* $\infty$*]*. Evaluation of *eval(plus(3, minus(5, 1)))* returns 7. Evaluated states are sets of relational symbols over constant symbols and complex structures, represented by constructor functions over constant symbols. For example, the relational symbol *on* in state description {*on([2 3* $\infty$*],* $p_1$*), on([*$\infty$*],* $p_2$*), on([1* $\infty$*],* $p_3$*)*} has constant arguments $p_1$, $p_2$, and $p_3$, and complex arguments *[2 3* $\infty$*]*, *[*$\infty$*]*, and *[1* $\infty$*]*. Relational symbols in $\mathcal{R}_\mathcal{C}$ can be considered as special terms, namely terms that evaluate to a truth value, also called boolean terms. In contrast, relations in $\mathcal{R}_\mathcal{P}$ are true if they match with the current state and false otherwise. In the following, we will speak of evaluation of expressions when referring to terms including such boolean terms.

**Definition 4.5 (Evaluation of Expressions over Ground Terms)** *An expression over* $\mathcal{L}(\mathcal{C}, \mathcal{F}, \mathcal{R}_\mathcal{C})$ *is evaluated in the following way:*

- $eval(c)$, $eval(f(t_1, \ldots, t_n))$ *as in def. 4.4.*
- *eval(*$r(t_1, \ldots, t_n)$*) = apply(r(eval(*$t_1$*), ... , eval(*$t_n$*))), for* $r \in \mathcal{R}_\mathcal{C}$ *and* $t_1, \ldots, t_n \in \mathcal{C} \cup \mathcal{F} \cup \mathcal{R}_\mathcal{C}$ *with eval(*$r(t_1, \ldots, t_n)$*)* $\in \{TRUE, FALSE\}$.

*For a set of atoms over* $\mathcal{L}(\mathcal{C}, \mathcal{F}, \mathcal{R}_\mathcal{C})$*, i. e., a conjunction of boolean expressions, evaluation is extended to:*

$$eval(\{r_1(\vec{t_1}), \ldots, r_n(\vec{t_n})\}) = \begin{cases} TRUE & if\ r_1(\vec{t_1}) = TRUE\ and \\ & \ldots\ and\ r_n(\vec{t_n}) = TRUE \\ FALSE & else. \end{cases}$$

For example, {*car([1 2 3* $\infty$*])* $\neq \infty$, *car([1 2 3* $\infty$*])* $<$ *car([*$\infty$*])*} evaluates to TRUE. In our planner, evaluation of expressions is realized by calling the meta-function *eval* provided by Common Lisp.

Before introducing goals and operators defined over $\mathcal{L}(\mathcal{C}, \mathcal{F}, \mathcal{R})$, we extend the definitions for substitution and matching (def. 2.3):

**Definition 4.6 (Product of Substitutions)** *The composition of substitutions* $\sigma \circ \sigma'$ *is the set of all compatible pairs* $(x \leftarrow t) \in \sigma$ *and* $(x' \leftarrow t') \in \sigma'$ *with* $x, x' \in \mathcal{X}$ *and* $t, t' \in \mathcal{L}(\mathcal{X}, \mathcal{C}, \mathcal{F})$. *Substitutions are compatible iff for each* $(x \leftarrow t) \in \sigma$ *there does not exist any* $(x' \leftarrow t') \in \sigma'$ *with* $x = x'$ *and* $t \neq t'$.
*The product of sets of substitutions* $\Sigma = \{\sigma_1, \ldots \sigma_n\}$ *and* $\Sigma' = \{\sigma'_1, \ldots \sigma'_m\}$ *is defined as pairwise composition* $\Sigma \cup \Sigma' = \sigma_i \circ \sigma'_j$ *for* $i = 1 \ldots n$, $j = 1 \ldots m$.

**Definition 4.7 (Matching and Assignment)** *Let $F = F_P \cup F_C$ be a formula with*

- $F_P = \{p_1(\vec{t_1}), \ldots, p_i(\vec{t_i})\} \in \mathcal{L}(\mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R}_\mathcal{P})$ *and*
- $F_C = \{r_1(\vec{t'_1}), \ldots, r_j(\vec{t'_j})\} \in \mathcal{L}(\mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R}_\mathcal{C})$, *and let*
- $A \in \mathcal{L}(\mathcal{C}, \mathcal{F}, \mathcal{R}_\mathcal{P})$ *be a set of evaluated atoms.*

*The set of substitutions $\Sigma$ for $F$ with respect to $A$ is defined as $\Sigma_P \cup \Sigma_C$ with $F_{P_{\sigma_i}} \subseteq A$ and* eval($F_{C_{\sigma_i}}$) = TRUE *for all $\sigma_i \in \Sigma$.*

- $\Sigma_P$ *is calculated by* match($F_P$, A) *as specified in definition 2.3.*
- $\Sigma_C$ *is calculated by* ass($\mathcal{X}_\mathcal{C}$) *for all free variables $x_1, \ldots, x_n = \mathcal{X}_\mathcal{C}(F_C)$ such that for all $\sigma_i \in \Sigma_C$ holds $\sigma_i = \{x_1 \leftarrow c_1, \ldots, x_n \leftarrow c_n\}$ where $c_i \in R(x_i)$ (constants $c_i$ in the range of variable $x_i$, see def. 4.2).*

Definition 4.7 extends matching of formulas with sets of atoms in such a way that only those matches are allowed where substitutions additionally fulfill the constraints.

Consider the state

$$A = \{on([2\ 3\ \infty], p_1), on([\infty], p_2), on([1\ \infty], p_3)\}.$$

Formula PRE of the *move* operator given in figure 4.1.b

$$F = \{on(?l_i, ?p_i), on(?l_j, ?p_j), car(?l_i) \neq \infty, car(?l_i) < car(?l_j)\}$$

can be instantiated to $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$ with

$$\sigma_1 = \{?p_i \leftarrow p_1,\ ?p_j \leftarrow p_2,\ ?l_i \leftarrow [2\ 3\ \infty],\ ?l_j \leftarrow [\infty]\},$$
$$\sigma_2 = \{?p_i \leftarrow p_3,\ ?p_j \leftarrow p_2,\ ?l_i \leftarrow [1\ \infty],\ ?l_j \leftarrow [\infty]\},$$
$$\sigma_3 = \{?p_i \leftarrow p_3,\ ?p_j \leftarrow p_1,\ ?l_i \leftarrow [1\ \infty],\ ?l_j \leftarrow [2\ 3\ \infty]\}$$

but not to

$$\sigma_4 = \{?p_i \leftarrow p_1, ?p_j \leftarrow p_3, ?l_i \leftarrow [2\ 3\ \infty], ?l_j \leftarrow [1\ \infty]\},$$
$$\sigma_5 = \{?p_i \leftarrow p_2, ?p_j \leftarrow p_1, ?l_i \leftarrow [\infty], ?l_j \leftarrow [2\ 3\ \infty]\},$$
$$\sigma_6 = \{?p_i \leftarrow p_2, ?p_j \leftarrow p_3, ?l_i \leftarrow [\infty], ?l_j \leftarrow [1\ \infty]\}.$$

A procedural realization for calculating all instantiations of a formula with respect to a set of atoms is to first calculate all matches and then modify $\Sigma$ by stepwise introducing the constraints. For the example above, we first calculate $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6\}$ considering only the sub-formula $F_P = \{on(?p_i, ?l_i), on(?p_j, ?l_j)\}$. Next, constraint $car(?l_i) \neq \infty$ is introduced, reducing $\Sigma$ to $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$. Finally, we obtain $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$ by applying the constraint $car(?l_i) < car(?l_j)$.

If the constraints contain free variables, each substitution in $\Sigma$ is combined with instantiations of these variables as specified in definition 4.6. An example for instantiating formulas with free variables is given in section 4.5.

**Definition 4.8 (Goal Representation)** *A goal $\mathcal{G}$ is a formula in $\mathcal{L}(\mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R})$.*

The goal given for *hanoi* in figure 4.1.b can alternatively be represented by:

$\{on([1\ 2\ 3\ \infty], p_3), on(?l_i, ?p_i), on(?l_j, ?p_j), \infty = car(?l_i), \infty = car(?l_j)\}$.

**Definition 4.9 (FPlan Operator)** *An FPlan operator op is described by preconditions (PRE), ADD and DEL lists, and updates (UP) with*
$PRE = PRE_P \cup PRE_C \in \mathcal{L}(\mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R}_\mathcal{P} \cup \mathcal{R}_\mathcal{C})$ *and* $ADD, DEL \in \mathcal{L}(\mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R}_\mathcal{P})$.
*An update UP is a list of function applications, specifying that a function $f(\vec{t}) \in \mathcal{L}(\mathcal{X}, \mathcal{C}, \mathcal{F})$ is applied to a term $t'_i$ which is argument of literal $p(\vec{t'})$ with $p \in \mathcal{R}_\mathcal{P}$.*

Updates *overwrite* the value of an argument of a literal. Realizing an update using ADD/DEL effects would mean to first calculate the new value of an argument, then deleting the literals in which this argument occurs and finally adding the literals with the new argument values. That is, modeling updates as specific process is more efficient. An example for an update effect is given in figure 4.1.b:

**change** $?l_i$ **in** $on(?l_i, ?p_i)$ **to** $cdr(?l_i)$.

defines that variable $?l_i$ in literal $on(?l_i, ?p_i)$ is updated to $cdr(?l_i)$. If more than one update effect is given in an operator, the updates are performed sequentially from the uppermost to the last update. An update effect cannot contain free variables, that is $\mathcal{X}(UP) \subseteq \mathcal{X}(PRE)$.

**Definition 4.10 (Updating a State)** *For a state $s = \{p_1(\vec{t_1}), \ldots, p_n(\vec{t_n})\}$ and an instantiated operator o with update effects $u \in UP$, updating is defined as replacement $t'_i := eval(f(\vec{t}))$ for a fixed argument $t'_i$ in a fixed literal $p_j(\vec{t'}) \in s$ with $\mathcal{X}(\vec{t}), \mathcal{X}(\vec{t'}) \subseteq \mathcal{X}(PRE)$.*
*For short we write $update(u, s)$.*
*Applying a list of updates $UP = (u_1, \ldots, u_n)$ to a state $s$ is defined as $update(UP, s) = update(u_n, update(u_{n-1}, \ldots, update(u_1, s)))$.*

For the initial state of *hanoi* in figure 4.1, *s = {on($p_1$, [1 2 3 $\infty$]), on($p_2$, [$\infty$]), on($p_3$, [$\infty$])}*, the update effects of the *move* operator can be instantiated to

$u_1$ = **change** $[\infty]$ **in** $on(p_3, [\infty])$ **to** *cons(car([1 2 3 $\infty$]) [$\infty$])*

$u_2$ = **change** [1 2 3 $\infty$] **in** $on(p_1, [1\ 2\ 3\ \infty])$ **to** $cdr([1\ 2\ 3\ \infty])$.

Applying these updates to $s$ results in:

$update((u_1, u_2), \{on(p_1, [1\ 2\ 3\ \infty]), on(p_2, [\infty]), on(p_3, [\infty])\}) =$

$update(u_2, \{on(p_1, [1\,2\,3\,\infty]), on(p_2, [\infty]), on(p_3, eval(cons(car([1\,2\,3\,\infty]), [\infty])))\} =$

$update(u_2, \{on(p_1, [1\ 2\ 3\ \infty]), on(p_2, [\infty]), on(p_3, [1\ \infty])\}) =$
$\{on(p_1, eval(cdr([1\ 2\ 3\ \infty]))), on(p_2, [\infty]), on(p_3, [1\ \infty])\} =$
$\{on(p_1, [2\ 3\ \infty]), on(p_2, [\infty]), on(p_3, [1\ \infty])\}.$

Note that the operator is fully instantiated before the update effects are calculated. The current substitution $\sigma \in \Sigma$ is *not* affected by updating. That is, if the value of an instantiated variable is changed, as for example $L_2 = [\infty]$ is changed to $L_2 = [1\ \infty]$, this change remains local to the literal specified in the update.

**Definition 4.11 ((Forward) Operator Application)** *For a state $s$ and an instantiated operator $o$, operator application is defined as $Res(o, s) = update(UP, s \setminus DEL(o) \cup ADD(o))$ if $PRE_P(o) \subseteq s$ and $eval(PRE_C(o)) = TRUE$.*

ADD/DEL effects are always calculated *before* updating. Examples for operators with combined ADD/DEL and update effects are given in section 4.

An FPlan *planning problem* is given as $P(\mathcal{O}, \mathcal{I}, \mathcal{G})$, where operators $\mathcal{O}$, initial states $\mathcal{I}$, and goals $\mathcal{G}$ are defined over the FPlan language, which extends Strips by allowing function applications. A plan for the *hanoi* problem specified in figure 4.1.b is given in figure 4.3.

# 3 EXTENSIONS OF FPLAN

## 3.1 BACKWARD OPERATOR APPLICATION

As described in section 1 in chapter 2, plan construction can be performed using either forward or backward search in the state space. To make FPlan applicable for backward planners, we introduce backward operator application.

Backward operator application for Strips (see def. 2.7) was defined so that an operator is backward applicable in a state $s$ if its ADD list can be matched with the current state, resulting in a predecessor state $s'$ where all elements of the ADD list are removed and which contains the literals of PRE and DEL. For operators containing update effects and constraints, the definition of backward operator application has to be extended with respect to applicability conditions and calculation of effects.

An operator is backward applicable if its ADD list can be matched with the current state $s$ *and* if those constraints that hold after forward operator application hold in $s$. We call such constraints the *postcondition* (POST) of an operator. In many domains, the constraints holding in a state after forward operator application are the inverse of the constraints specified in the operator precondition. For example, for the *hanoi* domain, the constraints for forward application are $\{car(?l_i) \neq \infty, car(?l_i) < car(?l_j)\}$ where $?l_i$ represents the disks lying on peg $?p_i$ and $?l_j$ represents the disks lying on peg $?p_j$ (see fig. 4.1.b). After executing *move*($?p_i$, $?p_j$), the "top" disk of $?l_i$ is inserted as head of list $?l_j$, and the constraints $\{car(?l_j) \neq \infty, car(?l_j) < car(?l_i)\}$ hold.

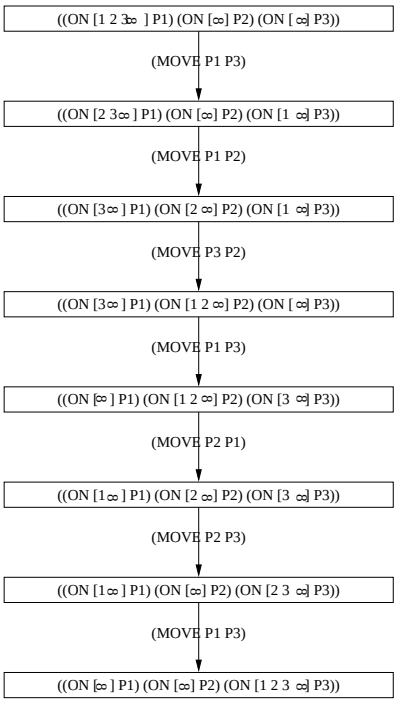



*Figure 4.3.* A Plan for *Tower of Hanoi*

The first constraint must hold, because a disk is moved to peg $?p_j$ and therefore $?l_j$ cannot be the empty peg represented as list *[∞]*. The second constraint must hold because all legal states in the *hanoi* domain contain stacks of disks with increasing size. Because $x < y$ holds for list $?l_i$ = *[x y ..]*, and because $x$ is the new head element of $?l_j$ after executing *move*, the new head element of list $?l_i$ ($y$) is larger than the new head element of list $?l_j$ ($x$).

To calculate a backward update effect, the inverse function $f^{-1}$ to update function $f$ must be known. In general, updating can involve a sequence of function applications. That is, $f = f_1 \circ \ldots \circ f_n$. To make backward plan *construction* compatible with forward plan *execution*, update effects must be restricted to bijective functions: $f(x) = y \leftrightarrow f^{-1}(y) = x$. For the *move* operator in the *hanoi* domain, the inverse function of removing the top element of a tower i. e. the head element of the list $?l_i$ and inserting it at as the head of $?l_j$ is to remove the first element of $?l_j$ and inserting it at as head of $?l_i$.

We do not propose an algorithm for automatic construction of correct constraints for backward application and inverse functions. Instead, we require that for each operator *op* its inverse operator $op^{-1}$ has to be defined in the domain specification. It is the responsibility of the author of an domain to guarantee

that $op^{-1}$ is the inverse operator of *op*; that is, that *Res(o,s')* = *s* holds for $Res^{-1}(o^{-1}, s) = s'$ for all operators.

When defining an inverse operator, the (inverse) precondition represents the conditions which must hold after forward application of the original operator – i. e. the inverse precondition corresponds to the original postcondition; the updates represent the inverse function as discussed above. The inverse operator $move^{-1}$ for *hanoi* is:

| | |
|---|---|
| **Operator:** | move$^{-1}$($?p_i$, $?p_j$) |
| **PRE:** | {on($?l_i$, $?p_i$), on($?l_j$, $?p_j$), car($?l_j$) $\neq \infty$, car($?l_i$) > car($?l_j$)} |
| **UPDATE:** | *change* $?l_i$ *in* on($?l_i$, $?p_j$) *to* cons(car($?l_j$), $?l_i$) |
| | *change* $?l_j$ *in* on($?l_j$, $?p_i$) *to* cdr($?l_j$) |

Because update effects are explicitly defined for inverse operator application, updates are performed as specified in definition 4.10.
For state $s = \{on([2\ 3\ \infty], p_1), on([\infty], p_2), on([1\ \infty], p_3)\}$ backward application of the instantiated operator $move^{-1}(p_1, p_3)$ results in $s = \{on([1\ 2\ 3\ \infty], p_1), on([\infty], p_2), on([\infty], p_3)\}$.

For operators defined with mixed ADD/DEL and update effects the inverse operator is also defined explicitely, with inverted precondition, ADD and DEL list. Application of an inverted operator can then be performed as specified in definition 4.11 – that is, backward operator application is replaced by forward application of the inverted operator. Examples for backward application of operators with combined ADD/DEL and update effects are given in section 4.

## 3.2 INTRODUCING USER-DEFINED FUNCTIONS

As defined in section 2, the set of function symbols $\mathcal{F}$ and the set of relational symbols $\mathcal{R}_\mathcal{C}$ of FPlan can contain arbitrary built-in and user-defined functions. When parsing a problem specification, each symbol which is part of Common Lisp and each symbol corresponding to the name of a user-defined function is treated as expression to be evaluated. Symbols in $\mathcal{R}_\mathcal{C}$ have to evaluate to a truth value.

An example specification of *hanoi* using built-in and user-defined functions is given in figure 4.4.[3] For the Towers of Hanoi domain the FPlan language is instantiated to $\mathcal{X} = \{?p_i, ?p_j, ?l_i, ?l_j\}$, $\mathcal{C}$ = {$p_1$, $p_2$, $p_3$, *1, 2, 3, nil*}, $\mathcal{F} = \{[\ ]^i, car^1, cdr^1, cons^2\}$, $\mathcal{R}_\mathcal{P} = \{on^2\}$, $\mathcal{R}_\mathcal{C}$ = {$not{-}empty^1$, $legal^2$}. The empty list *[ ]* corresponds to *nil*. The user-defined functions can be defined over additional built-in and user-defined functions which are not included in $\mathcal{L}$. For a Lisp-implemented planner – like our system DPlan – after reading in a

[3] For our planning system, functions are defined in Lisp syntax, for example: `(defun legal (l1 l2) (if (null l2) T (< (car l1) (car l2))))`.

| | |
|---|---|
| **Operator:** | move($?p_i$, $?p_j$) |
| **PRE:** | {on ($?l_i$, $?p_i$), on($?l_j$, $?p_j$), not-empty($?l_i$), legal($?l_i$, $?l_j$)} |
| **UPDATE:** | *change* $?l_j$ *in* on($?l_j$, $?p_j$) *to* cons(car($?l_i$), $?l_j$) |
| | *change* $?l_i$ *in* on($?l_i$, $?p_i$) *to* cdr($?l_i$) |
| **Goal:** | {on([ ], $p_1$), on([ ], $p_2$), on([1 2 3], $p_3$)} |
| **Initial State:** | {on([1 2 3], $p_1$), on([ ], $p_2$), on([ ], $p_3$)} |
| **Functions:** | not-empty($l$) = not(null($l$)) |
| | legal($l_1$, $l_2$) = *if* null($l_2$) *then* TRUE *else* $<$ (car($l_1$), car($l_2$)) |

*Figure 4.4.* *Tower of Hanoi* with User-Defined Functions

problem specification, all declared user-defined functions are appended to the set of built-in Lisp functions and interpreted in the usual manner.

# 4 EXAMPLES

In the previous sections, we presented FPlan with a functional version of the *hanoi* domain, demonstrating how operators with ADD/DEL effects can be alternatively modeled with update effects. In the following, we will give a variety of examples for problem specifications with FPLan. First we will show how problems involving resource constraints can be modeled. Afterwards, we will give examples for a numerical domain and domain specifications involving operators with mixed ADD/DEL effects and updates. We will show how standard programming problems can be modeled in planning, and finally we will present preliminary ideas for combining planning with constraint logic programming.

For selected problems we include empirical comparisions, demonstrating that plan construction with function updates is significantly more efficient than plan construction using ADD/DEL effects. We tested all examples with our backward planner DPlan (Schmid and Wysotzki, 2000b). Therefore, we present standard forward operator definitions together with the inverse operators as specified in section 3.1.

## 4.1 PLANNING WITH RESOURCE VARIABLES

Planning with resource constraints can be modeled as a special case of function updates with FPlan. As an example we use an airplane domain (Koehler, 1998). A problem specification in FPlan is given in figure 4.5.

The operator *fly* specifies what happens when an airplane $?plane$ flies from airport $?x$ to airport $?y$. We consider two resources: the amount of fuel and the amount of time the plane needs to fly from one airport to another. In the initial state the tank is filled to capacity (*fuel–resource(750))* and no time has

| | |
|---|---|
| **Operator:** | **fly**($?plane$, $?x$, $?y$) |
| **PRE:** | {at($?plane$, $?x$), fuel–resource($?plane$, $?fuel$), $?fuel \geq$ distance($?x$, $?y$)/3} |
| **ADD:** | {at($?plane$, $?y$)} |
| **DEL:** | {at($?plane$, $?x$)} |
| **UPDATE:** | *change* $?fuel$ *in* fuel–resource($?plane$, $?fuel$) *to* calc–fuel($?fuel$, $?x$, $?y$)<br>*change* $?time$ *in* time–resource($?plane$, $?time$) *to* calc–time($?time$, $?x$, $?y$) |
| **Goal:** | {at(p1, berlin)} |
| **Initial State:** | {at(p1, london), fuel–resource(p1, 750), time–resource(p1, 0)} |
| **Functions:** | calc–fuel($?fuel$, $?x$, $?y$) = $?fuel -$ distance($?x$, $?y$)/3<br>calc–time($?time$, $?x$, $?y$) = $?time + 3/20 *$ distance($?x$, $?y$) |

*Figure 4.5.* A Problem Specification for the *Airplane* Domain

*Table 4.1.* Database with Distances between Airports

| | Berlin | Paris | London | New York |
|---|---|---|---|---|
| Berlin | – | 540 m | 570 m | 3960 m |
| Paris | 540 m | – | 210 m | 3620 m |
| London | 570 m | 210 m | – | 3460 m |
| New York | 3960 m | 3620 m | 3460 m | – |

yet been spent (*time–resource(0)*). A resource constraint that must hold before the action of flying the plane from one airport $?x$ to another airport $?y$ can be carried out would be: *at(*$?plane$, $?x$*), (?fuel* $\geq$ *distance(*$?x$, $?y$*)/3)*. That is the plane has to be at airport $?x$ and there has to be enough fuel in the tank to travel the distance from $?x$ to $?y$.

The distance between two airports is obtained by calling the function *distance*(?x, ?y) which returns the distance value by consulting an underlying database (see table 4.1). The distances can alternatively be modeled as static predicates (for example *distance(berlin, paris, 540)*, etc.). Using a database query function is more efficient than modeling the distances with static predicates because static predicates have to be encoded in the current state and finding a certain distance (instantiating the literal) requires matching which takes considerably longer than a database query.

The position of the plane $?plane$ is modeled with the literal *at*. When flying from $?x$ to $?y$ the literal *at(*$?plane$, $?x$*)* is deleted from and the literal *at(*$?plane$, $?y$*)* is added to the set of literals describing the current state. The resource variable $?fuel$ described by the relational symbol *fuel–resource* is updated with the result of the user-defined function *calc–fuel* which calculates the consumption of fuel according to the traveled distance. The resource

| | |
|---|---|
| **Operator:** | **fly**$^{-1}$(?*plane*, ?$x$, ?$y$) |
| **PRE:** | {at(?*plane*, ?$y$), time–resource(?*plane*, ?*time*), fuel–resource(?*plane*, ?*fuel*), ?*time* $\geq$ distance(?$x$, ?$y$)) $*$ 3/20) } |
| **ADD:** | {at(?*plane*, ?$x$)} |
| **DEL:** | {at(?*plane*, ?$y$)} |
| **UPDATE:** | *change* ?*fuel* *in* fuel–resource(?*plane*, ?*fuel*) *to* calc–fuel$^{-1}$(?*fuel*, ?$x$, ?$y$)<br>*change* ?*time* *in* time–resource(?*plane*, ?*time*) *to* calc–time$^{-1}$(?*time*, ?$x$, ?$y$) |
| **Functions.:** | calc–fuel$^{-1}$(?*fuel*, ?$x$, ?$y$) = ?*fuel* + distance(?$x$, ?$y$)/3 \| 750<br>; 750 is the maxium capacity of the tank<br>calc–time$^{-1}$(?*time*, ?$x$, ?$y$) = ?*time* $-$ 3/20 $*$ distance(?$x$, ?$y$) |

*Figure 4.6.* Specification of the Inverse Operator *fly*$^{-1}$ for the *Airplane* Domain

variable ?*time* described by the relational symbol *time–resource* is updated in a similar way asuming that it takes $3/20 * distance(?x, ?y)$ to fly the distance from airport ?$x$ to ?$y$.

When ADD/DEL and update effects occur together in one operator the update effect is carried out on the state obtained after calculating the ADD/DEL effect (definition 4.11). For backward planning we have to specify a corresponding inverse operator *fly*$^{-1}$ (see fig. 4.6). The precondition for *fly*$^{-1}$ requests that the plane be at airport ?$y$ and you have more time left than it takes to fly the distance between airport ?$x$ and airport ?$y$. The resource variables *?fuel* and *?time* are updated with the result of the inverse functions (*calc–fuel*$^{-1}$ and *calc–time*$^{-1}$).

Updating of state variables as proposed in FPlan is more flexible than handling resource variables separately (as in Koehler (1998)). While in Koehler (1998), *fuel* and *time* are global variables, in FPlan the current value of fuel and time are arguments of relational symbols *fuel–resource(?plane, ?fuel)*, *time–resource(?plane,?fuel)*. Therefore, while modeling a problem involving more than one plane can be easily done in FPlan this is not possible with Koehler's approach. For example we can specify the following goal and initial state:

| | |
|---|---|
| **Goal:** | {at(p1, berlin), at(p2, paris)} |
| **Initial State:** | {at(p1, berlin), fuel–resource(p1, 750), time–resource(p1, 0),<br>at(p2, paris), fuel–resource(p2, 750), time–resource(p2, 0)} |

Modeling domains with time or cost resources is simple when function applications are allowed. Typical examples are job scheduling problems – for example the machine shop domain presented in (Veloso et al., 1995). To model time steps, a relational symbol *time(?t)* can be introduced. Time ?$t$ is initially zero and each operator application results in ?$t$ being incremented by one step. To model a machine that is occupied during a certain time interval, a relational symbol *occupied(?m, ?o)* can be used where ?$m$ represents the name of a

| | |
|---|---|
| **Operator:** | **pour**(?$j1$, ?$j2$) |
| **PRE:** | {volume(?$j1$, ?$v1$), volume(?$j2$, ?$v2$), capacity(?$j2$, ?$c2$)) (?$v2 <$ ?$c2$), (?$v1 > 0$)} |
| **UPDATE:** | *change* ?$v1$ *in* volume(?$j1$, ?$v1$) *to* $max(0,$ ?$v1-$?$c2+$?$v2$) *change* ?$v2$ *in* volume(?$j2$, ?$v2$) *to* $min($?$c2,$ ?$v1+$?$v2$) |
| **Operator:** | **pour**$^{-1}$(?$j1$, ?$j2$) |
| **PRE:** | {volume(?$j1$, ?$v1$), volume(?$j2$, ?$v2$), capacity(?$j2$, ?$c2$)) ((?$v2$ = ?$c2$) or (?$v1$ = 0))} |
| **UPDATE:** | *change* ?$v1$ *in* volume(?$j1$, ?$v1$) *to* $max(0,$ ?$v1-$?$c2+$?v2) *change* ?$v2$ *in* volume(?$j2$, ?$v2$) *to* $min($?$c2,$ ?$v1+$?$v2$) |
| **Statics:** | {capacity(a, 36), capacity(b, 45), capacity(c, 54)} |
| **Goal:** | {volume(a, 25), volume(b, 0), volume(c, 52)} |
| **Initial State:** | {volume(a, 16), volume(b, 7), volume(c, 34) } |

*Figure 4.7.* A Problem Specification for the *Water Jug* Domain

machine and ?$o$ the last time slot where it is occupied with ?$o = 0$ representing that the machine is free to be used. For each operator involving the usage of a machine, a precondition requesting that the machine is free for the current time slot can be introduced. If a machine is free to be used, the *occupied* relation is updated by adding the current time and the amount of time steps the executed action requests. It can also be modeled that occupation time does not only depend on the kind of action performed but also on the kind of object involved (e. g., polishing a large object could need three time steps, while polishing a small object needs only one time step).

## 4.2 PLANNING FOR NUMERICAL PROBLEMS – THE WATER JUG DOMAIN

Numerical domains as the *water jug* domain presented by Pednault (1994), cannot be modeled with a Strips-like representation. Figure 4.7 shows an FPlan specification of the *water jug* domain: We have three jugs of different volumes and different capacities. The operator *pour* models the action of pouring water from one jug *?j1* into another jug *?j2* until either *?j1* is empty or *?j2* is filled to capacity. The capacities of the jugs are statics while the actual volumes are fluents. The resulting volume *?v1* for jug *?j1* is either zero or what remains in the jug when pouring as much water as possible into jug *?j2*: max$[0, v1 - c2 + v2]$. The resulting volume *?v2* for jug *?j2* is the either its capacity or its previos volume plus the volume of the first jug *?j1*: min$[c2, v1 + v2]$.

After pouring water from one jug into another the postcondition *(?$v2$ = ?$c2$) or (?$v1$ = 0)* must hold. That is the volume ?$v1$ of the first jug *?j1* must be zero or the volume *?v2* of the second jug *?j2* must equal its capacity

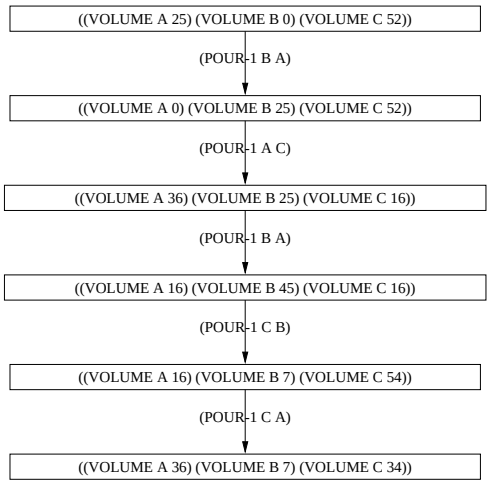



*Figure 4.8.* A Plan for the *Water Jug* Problem

*?c2*. When specifying the inverse operator $pour^{-1}$ for backward planning this postcondition becomes the precondition of $pour^{-1}$. The update functions are inverted: The volume *?v1* of the first jug *?j1* is updated with the result of $min[?c1, ?v1+?v2]$ and the volume *?v2* of the second jug *?v2* is updated with the result of $max[0, ?v2-?c1+?v1]$. The resulting plan for the given problem specification is shown in figure 4.8.

For this numerical problem there exists no backward plan for all initial states which can be transformed into the goal by forward operator application. For instance, for the initial state *volume(a, 16), volume(b, 27), volume(c, 34)* the following action sequence results in a goal state:

| | |
|---|---|
| *volume(a, 16), volume(b, 27), volume(c, 34)* | *pour(c, b)* |
| *volume(a, 16), volume(b, 45), volume(c, 16)* | *pour(b, a)* |
| *volume(a, 36), volume(b, 25), volume(c, 16)* | *pour(a, c)* |
| *volume(a, 0), volume(b, 25), volume(c, 52)* | *pour(b, a)* |
| *volume(a, 25), volume(b, 0), volume(c, 52)* | |

but the initial state is not a legal predecessor of *volume(a, 16), volume(b, 45), volume(c, 16)* when applying the $pour^{-1}$ operator. That is, backward planning is incomplete! The reason for this incompleteness is that the initial state in forward planning can be any arbitrary amount of water in the jugs, while for backward planning is has to be a state obtainable by operator application. A remedy for this problem is to introduce a second operator *fill–jugs(?j1, ?j2, ?j3)* which has no precondition. This operator models the initial filling up of jugs from a water source.

*Table 4.2.* Performance of FPlan: *Tower of Hanoi*

| Tower of Hanoi | | | |
|---|---|---|---|
| Number of disks | 3 | 4 | 5 |
| States | 27 | 81 | 243 |
| Performance gain | 96.3% | 98.7% | 99.8% |

## 4.3 FUNCTIONAL PLANNING FOR STANDARD PROBLEMS – TOWER OF HANOI

A main motivation for modeling operator effects with updates of state variables by function application given by Geffner (2000) is that plan construction can be performed more efficiently. We give an empirical demonstration of this claim by comparing the running times when planning for the standard specification with those of the the functional specification of Tower of Hanoi (see fig. 4.1). The inverse operator $move^{-1}$ was specified in section 3.1.

The running times were obtained with our system DPlan on an Ultra Sparc 5 system. DPlan is a universal planner implemented in Lisp. Because DPlan is based on a breadth-first strategy and because we did not focus on efficiency – for example we did not use OBDD representations (Cimatti et al., 1998) – it has longer running times than state of the art planers. Since we are interested in comparing the performance when planning for the different specifications we do not give the absolute times but the performance gain in percent, calculated as $[st - ft * 100]/st$, where $st$ is the running time for the standard specification and $ft$ the running time for the functional specification (see table 4.2).[4]

Because in the standard representation the relations between the sizes of the different discs must be specified explicitely by static relations, the number of relational symbols is quite large (18 literals for three disks, 25 for four disks, etc.). In the functional representation, built-in functions can be called to determine wether a disk of the size 2 is smaller than a disk of the size 3. The number of literals is very small (3 literals for an arbitrary number of disks). Consequently planning for the standard representation requires more matching than planning for the functional one, where the constraint of the precondition has to be solved instead.

[4]The absolute values are (standard/functional): 3 discs: 34.5 sec/1.26 sec; 4 discs: 335.89 sec/4.47 sec; 5 discs: 10015.59 sec/19.3 sec; 6 discs (729 states): $>$15 h/62.28 sec.

| | |
|---|---|
| **(a) Operator:** | **clearblock**($?x$) |
| **PRE:** | {$?y$ = topof($?x$), clear($?y$)} |
| **ADD:** | {clear($?x$)} |
| **DEL:** | {on($?y$,$?x$)} |
| **UPDATE:** | *change* $?block$ *in* LastBlockMoved($?block$) *to* $?y$ |
| **(b) Operator:** | **puttable**($?x$) |
| **PRE:** | {$?x$ = topof($?y$), clear($?x$)} |
| **ADD:** | {clear($?x$)} |
| **DEL:** | {on($?x$,$?y$)} |
| **UPDATE:** | *change* $?block$ *in* LastBlockMoved($?block$) *to* $?x$ |
| **(c) Operator:** | **put**($?x$, $?y$) |
| **PRE:** | {clear($?x$), clear($?y$) } |
| **ADD:** | {on($?y$,$?x$)} |
| **DEL:** | {clear($?x$)} |
| | **WHEN** on($?x$ $?z$) **ADD** {on($?x$ ?y), clear($?z$)} |
| | **DEL** {clear($?y$), on($?x$ $?z$)} |
| **UPDATE:** | *change* $?block$ *in* LastBlockMoved($?block$) *to* $?y$ |
| **Functions:** | topof($?x$) = if on($?y$,$?x$) then $?y$ else nil |

*Figure 4.9.* Blocks-World Operators with Indirect Reference and Update

## 4.4 MIXING ADD/DEL EFFECTS AND UPDATES – EXTENDED BLOCKSWORLD

In this section we present an extended version of the standard blocks-world (see fig. 4.9). First we introduce an operator *clearblock* which clears a block *?x* by putting the block lying on top of *?x* on the table – if block *topof(?x)* is clear. That is, a block is addressed in an indirect manner as result of a function evaluation (Geffner, 2000). As terms can be nested this corresponds to an infinite type. For a state where *on(A, B)* holds, the function *topof(B)* returns $A$, while for a state where *on(C, B)* holds, *topof(B)* returns $C$. The standard blocks-world operators *put* and *puttable* are also formulated using the *topof* function. Furthermore all operators are extended by an update for the value of the *LastBlockMoved* (Pednault, 1994). Note that we used a conditional effect for specifying the *put* operator. Our current implementation allows conditioned effects for ADD/DEL effects but not for updates.

The update effect changing the value of *LastBlockMoved* cannot be handeled in backward planning because we cannot define inverse operators including an inverse update of the argument *LastBlockMoved*. The inverse function would have to return the block that will be moved in the next step. At the current time step we do not know which block this will be.

**(a) Standard representation**

| | |
|---|---|
| **Operator:** | **swap**($?i$, $?j$, $?x$) |
| **PRE:** | {is–at($?x$, $?i$), is–at($?y$, $?j$), greater($?x$, $?y$) |
| **ADD:** | {is–at($?x$, $?j$), is–at($?y$, $?i$)} |
| **DEL:** | {is–at($?x$, $?i$), is–at($?y$, $?j$)} |
| | |
| **Initial State:** | {is–at(3, p1), is–at(2, p2), is–at(1, p3)} |
| **Goal:** | {is–at(1, p1), is–at(2, p2), is–at(3, p3)} |

**(b) Functional representation**

| | |
|---|---|
| **Operator:** | **swap**($?i$, $?j$, $?x$) |
| **PRE:** | {slist($?x$), nth($?i$, $?x$) > nth($?j$, $?x$)} |
| **UPDATE:** | *change* $?x$ *in* slist($?x$) *to* swap($?i$, $?j$, $?x$) |
| | |
| **Variables:** | {($?i$ :range (0 2)) ($?j$ :range (0 2))} |
| **Initial State:** | {slist([ 3 2 1 ])} |
| **Goal:** | {slist([ 1 2 3 ])} |
| **Functions:** | swap ($?i$, $?j$, $?x$) = (let ((temp (nth j x))) (setf (nth j x) (nth i x) (nth i x) temp) x)) |

*Figure 4.10.* Specification of *Selection Sort*

Other extended blocks-world domains can be modeled with FPlan. For example, inluding a robot agent who stacks and unstacks blocks with an associated energy level which decreases with each action it performes The robot's energy can be described with the relational symbol *energy(?robot, ?energy)* and is consumed according to the weight of the actual block being carried by updating the variable *?energy* in *energy(?robot, ?energy)*.

## 4.5 PLANNING FOR PROGRAMMING PROBLEMS – SORTING OF LISTS

With the extension to functional representation a number of programming problems for example list sorting algorithms such as bubble, merge or selection sort can be planned more efficiently. We have specified selection sort in the standard and the functional way (see fig. 4.10).

In the functional representation we can use the built-in function "$>$" instead of the predicate *greater*, the constructor for lists *slist*, and the function *nth(n, L)* to reference the nth element of the list $L$ instead of specifying the position of each element in the list by using the literal *is–at(?x, ?i)*. In the functional representation the indices to the list *?i* and *?j* are free variables, which – for lists with three elements – can range from zero to two. When calculating the possible instantiations for the operator *swap* the variables *?i, ?j* can be assigned

*Table 4.3.* Performance of FPlan: *Selection Sort*

| selection sort | | | |
|---|---|---|---|
| Number of list elements | 3 | 4 | 5 |
| States | 6 | 24 | 120 |
| Performance gain | 27.3% | 81.3% | 96.1% |

any value within their range for which the precondition holds (definition 4.2). The function swap is defined as an external function. The inverse function to swap is the function swap itself.

The performance gain when planning for the functional representation is large (table 4.3) but not as stunning as in the Hanoi domain.[5] This may be due to the free variables $?i$ and $?j$ that have to be assigned a value according to their range. Consequently the constraint solving takes some longer but is still faster than the matching of the literals for the standard version (6 for three list elements, 10 for four, etc. versus only one literal for the functional representation).

## 4.6 CONSTRAINT SATISFACTION AS SPECIAL CASE OF PLANNING

Logical programs can be defined over rules and *facts* (Sterling and Shapiro, 1986). Rules correspond to planning operators. Facts are similar to static relations in planning, that is, they are assumed to be true in all situations. Constraint logic programming extends standard logic programming so that constraints can be used to efficiently restrict variable bindings (Frühwirth and Abdennadher, 1997). In figure 4.11 an example for a program in constraint Prolog is given. The problem is to compose a light meal consisting of an appetizer, a main dish and a dessert which all together contains not more than a given calorie value (see Frühwirth and Abdennadher, 1997).

The same problem can be specified in FPlan (figure 4.12). The goal consisting of a conjunction of literals now contains free variables and a set of constraints determining the values of those variables. In this case $?x$, $?y$, $?z$ are free variables that can be assigned any value within their range (here enumerations of dishes). The ranges of the variables $?x, ?y, ?z$ are specified for the domain and not for a problem. The function *calories* looks up the calorie value of a dish in an association list.

[5]The absolute values are (standard/functional): 3 elem.: 0.33 sec/0.24 sec; 4 elem.: 8.93 sec/1.67 sec; 5 elem.: 547.05 sec/21.57 sec; 6 elem. (720 states): $>$15 h/717.41 sec.

```
lightmeal(A, M, D)  ←  1400 ≥ I + J + K, I > 0, J > 0, K > 0,
                       appetizer(A, I), main(M, J), dessert(D, K).

appetizer(radishes, 50).   main(beef, 1000).    dessert(icecream, 600).
appetizer(soup, 300).      main(pork, 1200).    dessert(fruit, 90).
```

*Figure 4.11.* *Lightmeal* in Constraint Prolog

| | |
|---|---|
| **Domain:** | Lightmeal |
| **variables:** | ($?x$ :range (radishes, soup) |
| | ($?y$ :range (beef, pork)) |
| | ($?z$ :range (fruit, icecream) |
| **Initial State:** | { } ; empty |
| **Goal:** | {(lightmeal($?x$, $?y$, $?z$), |
| | (calories($?x$) + calories($?y$) + calories($?z$))≤ 1400} |
| **Functions:** | calories($?dish$) = cadr(assoc($?dish$, calorie-list)) |
| | calorie-list := ((radishes 50) (beef 1000) (fruit 90) |
| | (icecream 600) (soup 300) (pork 1200) ...) |

*Figure 4.12.* Problem Specification for *Lightmeal*

There are no operators specified for this example. The planning process in this case solves the goal constraint by finding those combinations of dishes that satisfy the constraint. For example a meal consisting of radishes with 50 calories, beef with 1000 calories and fruit with 90 calories.

Chapter 5

# CONCLUSIONS AND FURTHER RESEARCH

*The first part of anything is usually easy.*
—Travis McGee in: John D. MacDonald, *The Scarlet Ruse*, 1973

As mentioned before, DPlan is intended as a tool to support the generation of finite programs. That is, plan generation is for small domains – with three or four objects – only. Generation of a universal plan covering all possible states of a domain, as we do with DPlan, is necessarily a complex problem because for most interesting domains the number of states grows exponentially with the number of objects. In the following, we first discuss what extensions and modifications would be needed to make DPlan competitive with state of the art planning systems. Afterwards, we discuss extensions which would improve DPlan as a tool for program synthesis. Finally, we discuss some relations to human problem solving.

## 1 COMPARING DPLAN WITH THE STATE OF THE ART

Effort of universal planning is typically higher than the average effort of standard state-based planning because it is based on breadth-first search. But, if one is interested in generating optimal plans, this price must be payed. Typically, universal planning terminates if the initial state is reached in some pre-image (see sect. 4.4 in chap. 2). In contrast, DPlan terminates after all states which can be transformed into the goal state are covered. Consequently, it is not possible to make DPlan more time efficient – for a domain with exponential growth an exponential number of steps is needed to enumerate all states. But DPlan could be made more memory efficient by representing plans as OBDDs (Jensen and Veloso, 2000).

To be applied to standard planning problems, it would be necessary to modify DPlan. The main problem to overcome would be to generate complete state descriptions by backward operator application from a set of top-level goals, that is, from an incomplete state description (see sect. 1 in chap. 3). Some recent work by Wolf (2000) deals with that problem: Complete state descriptions are constructed as maximal sets of atoms which are not mutually excluded by the operator definitions of the domain.

Furthermore, DPlan currently terminates with the universal plan. For standard planning, a functionality for extracting a linear plan for a fixed state must be provided. Plan extraction can be done for example by depth-first search in the universal plan (Schmid, 1999).

In the context of planning and control rule learning (see sect. 5.2 in chap. 2), we argued that planning can profit from program synthesis: First, a universal plan is constructed for a domain with a small number of objects, then a recursive rule for the complete domain (with a fixed goal, such as "build a tower of sorted blocks") is generalized. Afterwards, planning can be omitted and instead the recursive rule is applied. To make this argument stronger, it is important to provide empirical evidence. We must show that plan generation by applying the recursive rule results in correct and optimal plans and that these plans are calculated in significantly less time than with a state-of-the-art planner. For that reason, DPlan must be extended such that (1) learned recursive rules are stored with a domain, and (2) for a new planning problem, the appropriate rule is extracted from memory and applied.[1]

## 2 EXTENSIONS OF DPLAN

To make it possible that DPlan can be applied to as large a set of problems which are of interest in program synthesis as possible, DPlan should work for a representation language with expressive power similar to PDDL (see sect. 2.1 in chap. 2). We already made an important step in that direction by introducing function application (chap. 4). The extension of the Strips planning language to function application has the following characteristics: Planning operators can be defined using arbitrary symbolical and numerical functions; ADD/DEL effects and updates can be combined; indirect reference to objects via function application allows for infinite domains; planning with resource constraints can be handled as special case.

The proposed language FPlan can be used to give function application in PDDL (McDermott, 1998b) a clear semantics. The described semantics of operator application can be incorporated in arbitrary Strips planning systems. In contrast to other proposals for dealing with the manipulation of state variables,

[1] Preliminary work for control rule application was done in a student project at TU Berlin, 1999.

such as ADL (Pednault, 1994), we do not represent them as specially marked "first class objects" (declared as fluents in PDDL) but as arguments of relational symbols. This results in a greater flexibility of FPlan, because each argument of a relational symbol may principally be changed by function application. As a consequence, we can model domains usually specified by operators with ADD/DEL effects alternatively by updating state variables (Geffner, 2000).

Work to be done includes detailed empirical comparisons of the efficiency of plan construction with operators with ADD/DEL effects versus updates and providing proofs of correctness and termination for planning with FPlan. FPlan currently has no restrictions to what functions can be applied. As a consequence, termination is not guaranteed. That is, we have to provide some restrictions on function application.

Currently, DPlan works on Strips extended by conditional effects and function application. For future extensions, we plan to include typing, negation in pre-conditions, and quantification. These extensions allow to generate plans for a larger class of problems. On the other hand, as already discussed for function application, each language extension results in a higher complexity for planning. While PDDL offers the syntax for all mentioned extensions, up to now only a limited number of work proposes an algorithmic realization (Penberthy and Weld, 1992; Koehler et al., 1997) and careful analyses of planning complexity are necessary (Nebel, 2000).

## 3 UNIVERSAL PLANNING VERSUS INCREMENTAL EXPLORATION

We argue, that for learning a recursive generalization for a domain, the complete structure of this domain must be known. Universal planning with DPlan results in a DAG which represents the shortest action sequences to transform each possible state over a fixed number of objects into a goal state. In chapter 8 we will show that from the structure of the DAG additionally information about the data type underlying this domain can be inferred which is essential for transforming the plan in a program term fit for generalization. Our approach is "all-or-nothing" – that is, either a generalization can be generated for a given universal plan or it cannot be generated and if a generalization can be generated it is guaranteed to generate correct and optimal action sequences for transforming all possible states into a goal state. A restriction of our approach is, that the planning goal must also be a generalization of the goal for which the original universal plan was generated (clear a block in a $n$ block tower, transport $n$ objects from $A$ to $B$, sort a list with $n$ elements, build a tower of $n$ sorted blocks, solve Tower of Hanoi for $n$ discs).

Other approaches to learning control rules for planning, namely learning in Prodigy (Veloso et al., 1995) and Geffner's approach (Martín and Geffner, 2000), on the one hand offer more flexibility but on the other hand cannot guar-

antee that learning in every case results in a better performance (see sect. 5.2 in chap. 2): The system is exposed incrementally to arbitrary planning experience. For example, in the blocks-world domain, the system might be first confronted with a problem "find an action sequence for transforming a state where all given four blocks $A$, $B$, $C$, and $D$ are lying on the table into one where $B$ is on $A$ and $C$ is on $D$; and next with a problem "find an action sequence for transforming a state where all given six blocks which are stacked in reverse order into one where blocks $A$ to $D$ are stacked into an ordered tower and $E$ and $F$ are lying on the table". Rules generalized from this experience might or might not work for new problems. For each new problem, the learned rules are applied and if rule application results in failure the system must extend or modify its learning experience.

This incremental approach to learning is similar to the models of human skill acquisition as, for example, proposed by Anderson (1983) in his ACT theory, although these approaches address only the acquisition of linear macros (see sect. 5.2 in chap. 2). To get some hints whether our universal planning approach has plausibility for human cognition, the following empirical study could be conducted: For a given problem domain, such as Tower of Hanoi, one group of subjects is confronted with all possible constellations of the three-disc problem and one group of subjects is confronted with arbitrary constellations of problems with different numbers of discs. Both groups have to solve the given problems. Afterwards, performance on arbitrary Tower of Hanoi problems is tested for both groups and both groups are asked after the general rule for solving Tower of Hanoi. We assume that explicit generation of all action sequences for a problem of fixed size, facilitates detection of the structure underlying a domain and thereby the formation of a general solution principle.

II

# INDUCTIVE PROGRAM SYNTHESIS: EXTRACTING GENERALIZED RULES FOR A DOMAIN

Chapter 6

# AUTOMATIC PROGRAMMING

*"So much is observation. The rest is deduction."*
—Sherlock Holmes to Watson in: Arthur Connan Doyle, *The Sign of Four*, 1930

Automatic programming is investigated in artificial intelligence and software engineering. The overall research goal in automatic programming is to automatize as large a part of the development of computer programs as possible. A more modest goal is to automatize or support special aspects of program development – such as program verification or generation of high-level programs from specifications. The focus of this chapter is on program generation from specifications – referred to as automatic program construction or program synthesis. There are two distinct approaches to this problem: Deductive program synthesis is concerned with the automatic derivation of correct programs from complete, formal specifications; inductive program synthesis is concerned with automatic generalization of (recursive) programs from *incomplete* specifications, mostly from input/output examples. In the following, we first (sect. 1) give an introductory overview of approaches to automatic programming, together with pointers to literature. Afterwards (sect. 2), we shortly review theorem proving and transformational approaches to deductive program synthesis. Since our own work is in the context of inductive program synthesis, we will go into more detail, presenting this area of research (sect. 3): In section 3.1, the foundations of automatic induction – that is, of machine learning – are introduced; grammar inference is discussed as theoretical basis of inductive synthesis. Afterwards, genetic programming (sect. 3.2), inductive logic programming (sect. 3.3), and the synthesis of functional programs (sect. 3.4) are presented. Finally, we discuss deductive versus inductive and functional versus

logical program synthesis (sect. 4). Throughout the text we give illustrations using simple programming problems.

# 1 OVERVIEW OF AUTOMATIC PROGRAMMING RESEARCH

## 1.1 AI AND SOFTWARE ENGINEERING

Software engineering is concerned with providing methodologies and tools for the development of software (computer programs). Software engineering involves at least the following activities:

**Specification:** Analysis of requirements and the desired behavior of the program and designing the internal structure of the program. Design specification might be stepwise refined, from an overall structure of software modules to the (formal) specification of algorithms and data structures.

**Development:** Realizing the software design in executable program code (programming, implementation).

**Validation:** Ensuring that the program does what it is expected to do. One aspect of validation – called **verification** – is that the implemented program realizes the specified algorithms. Program verification is realized by giving (formal) proofs that the program fulfills the (formal) specification. A verified program is called *correct* with respect to a specification. The second aspect of validation is that the program meets the initially specified requirements. This is usually done by testing.

**Maintenance:** Fixing program errors, modifying and adding features to a program (updating).

Software products are expected to be correct, efficient, and transparent. Furthermore, programs should be easily modifiable, maintaining the quality standards correctness, efficiency, and transparency. Obviously, software development is a complex task and since the eighties a variety of computer-aided software engineering (CASE) tools have been developed – supporting project management, design (e. g., checking the internal consistency of module hierarchies), and code generation for simple routine tasks (Sommerville, 1996).

### 1.1.1 KNOWLEDGE-BASED SOFTWARE ENGINEERING

A more ambitious approach is knowledge-based software engineering (KBSE). The research goal of this area is to support all stages of software development by “intelligent” computer-based assistants (Green et al., 1983). KBSE and automatic programming are often used as synonyms.

A system providing intelligent support for software development must include several aspects of knowledge (Smith, 1991; Green and Barstow, 1978):

General knowledge about the application domain and the problem to be solved, programming knowledge about the given domain, as well as general programming knowledge about algorithms, data structures, optimization techniques, and so on. AI technologies, especially for knowledge representation, automated inference, and planning, are necessary to build such systems. That is, an KBSE system is an expert system for software development.

The lessons learned from the limited success of expert systems research in the eighties, also apply to KBSE: It is not realistic to demand that a single KBSE system might cover all aspects of knowledge and all stages of software development. Instead, systems typically are restricted in at least one of the following ways (Flener, 1995; Rich and Waters, 1988):

- Constructing systems for expert specifiers instead of end-users.
- Restricting the system to a narrow domain.
- Providing interactive assistance instead of full automatization.
- Focusing on a small part of the software development process.

Examples for specialized systems are *Aries* (Johnson and Feather, 1991) for specification acquisition, KIDS (Smith, 1990) for program synthesis from high-level specifications (see sect. 2.2.3), or PVS (Dold, 1995) for program verification. Automatic programming is mainly an area of basic research. Up to now, only a small number of systems are applicable to real-world software engineering problems.

Below, we will introduce approaches to program synthesis, that is KBSE systems addressing the aspect of code generation from specifications, in detail. From a software engineering standpoint, the main advantage of automatic code generation is, that ex-post verification of programs becomes obsolete if it is possible to automatically derive (correct) programs from specifications. Furthermore, the problems of program modification can be shifted to the more abstract – and therefore hopefully more transparent – level of specification.

### 1.1.2 PROGRAMMING TUTORS

KBSE research aims at providing systems that support expert programmers. Another area of research, where AI and software engineering interact, is the development of tutor systems for support and education of student programmers. Such tutoring systems must incorporate programming knowledge to a smaller extend than KBSE systems, but additionally, they have to rely on knowledge about efficient teaching methods. Furthermore, user-modeling is critical for providing helpful feedback for programming errors.

Examples for programming tutors are the Lisp-tutors from Anderson, Conrad, and Corbett (1989) and Weber (1996) and the tutor *Proust* from Johnson

(1988). Anderson's Lisp-tutor is based on his ACT theory (Anderson, 1983). The tutoring strategy is based on the assumption that programming is a cognitive skill, based on a growing set of production rules. Training focuses on acquisition of such rules ("IF the goal is to obtain the first element of a list $l$ THEN write $(car\ l)$"), and errors are corrected immediately to avoid that students acquire faulty rules. Error-recognition and feedback is based on a library of expert rules and rules which result in programming errors. The later were acquired in a series of empirical studies of novice programmers. A faulty program is tried to be reproduced by applying rules from the library and feedback is based on the rules which generated the errors. Weber's tutor is based on episodic user modeling. The programs a student generates over a curriculum are stored as schemes and in a current session the stored programming episodes are used for feedback. The *Proust* system is based on the representation of plans (corresponding to high-level specifications of algorithms). Ideally, more than one correct program can be derived from a correct plan. Errors are detected by trying to identify the faulty plan underlying a given program.

Programming tutors were mainly developed during the eighties. Simultaneously, novice programmers were studied intensively in cognitive psychology (Widowski and Eyferth, 1986; Mayer, 1988; Soloway and Spohrer, 1989). One reason for this interest was that this area of research offered an opportunity to bring theories and empirical results of psychology to application; a second reason was that programming offers a relatively narrow domain which does not strongly depend on previous experience in other areas to study the development of human problem solving skills (Schmid, 1994; Schmid and Kaup, 1995).

## 1.2 APPROACHES TO PROGRAM SYNTHESIS

Research in program synthesis addresses the problem of automatic generation of program code from specifications. The nature of this problem depends on the form in which a specification is given. In the following, we first introduce different ways to specify programming problems. Afterwards we introduce different synthesis methods. Synthesis methods can be divided in two classes – deductive program synthesis from complete formal specifications and inductive program synthesis from incomplete specifications. Therefore, we first contrast deductive and inductive inference, and then characterize deductive and inductive program synthesis as special cases of these inference methods. Finally, we will mention schema-based and analogy-based approaches as extensions of deductive and inductive synthesis.

### 1.2.1 METHODS OF PROGRAM SPECIFICATION

Table 6.1 gives an illustration of possible ways in which a program for returning the last element of a non-empty list can be specified.

*Table 6.1.* Different Specifications for *Last*

| | |
|---|---|
| **Informal Specification** | |
| *Return the last element of a non-empty list.* | |
| **Declarative Programs** | |
| `last([X]) :- X.`<br>`last([X\|T]) :- last(T).` | (logic program, Prolog) |
| `fun last(l) =`<br>`if null(tl(l)) then hd(l) else last(tl(l));` | functional program (ML) |
| `fun last (x::nil) = x`<br>`\| last(x::rest) = last(rest);` | (ML with pattern matching) |
| **Complete, Formal Specifications** | |
| *last(l)* $\Leftarrow$ *find z such that for some y, l = y* $\circ$ *[z]*<br>*where islist(l) and l* $\neq$ *[ ]* | (Manna & Waldinger, 1992, p. 4) |
| `last :  seq_1 X --> X`<br>`forall s :  seq_1 X • last s = s(#s)` | (Z)<br>(Spivey, 1992, p. 117) |
| **Incomplete Specifications** | |
| `last([1], 1), last([2 5 6 7], 7), last([9 3 4], 4)` | (I/O Pairs, logical) |
| `last([1]) = 1, last([2 7]) = 7, last([5 3 4]) = 4` | (I/O Pairs, functional, first 3 inputs) |
| `last([`$x_1$`]) =` $x_1$`, last([`$x_1$ $x_2$`]) =` $x_2$`, last([`$x_1$ $x_2$ $x_3$`]) =` $x_3$ | (Generic I/O Pairs) |
| `last([1 2 3])` $\hookrightarrow$ `last([2 3])` $\hookrightarrow$ `last([3])` $\hookrightarrow$ `3` | (Trace) |
| `last([l]) = if null(tl(l)) then hd(l) else if null(tl(tl(l)))`<br>`then hd(tl(l)) else if null(tl(tl(tl(l)))) then hd(tl(tl(l)))` | (Complete generic trace) |

If a specification is given just as an *informal (natural language) description of requirements* (Green and Barstow, 1978), we are back at the ambitious goal of automatic programming – often ironically called "auto*magic* programming" (Rich and Waters, 1986a, p. xi). Besides the problem of automatic natural language understanding in general, the main difficulty of such an approach would be to derive the required information – for example, the desired input/output relation – from an informal statement. For the *last* specification in table 6.1, the synthesis system must have knowledge about lists, what it means that a list is not empty, what "element of a list" refers to and what is meant by "last".

If a specification is given as a *complete formal representation of an algorithm*, we have a more realistic goal, which is addressed in approaches of deductive program synthesis. Both example specifications in table 6.1 give

a non-operational description of the problem to be solved. The specification given by Manna and Waldinger (1992) states that for a non-empty list $l$ there must be a sub-list $y$ such that $y$ concatenated with the searched for element $z$ is equal to $l$. The $Z$ specification (Spivey, 1992) states that for all non-empty sequences $s$ (defined by $seq_1\ X$) *last* returns the element on the last position of this list ($s(\#s)$) where $\#s$ gives the length of $s$.

The distinction between "very high level" specification languages – such as *Gist* or *Z* (Potter, Sinclair, and Till, 1991) – and "high level" programming languages is not really strict: What is seen as specification language today might be a programming language in the future. In the fifties, assembler languages and the first compiler language *Fortran* where classified as automatic programming systems (see Rich and Waters, 1986a, p. xi, for a reprint from Communications of the ACM, Vol. 1(4), April 1958, p. 8).

Formal specification languages as well as programming languages allow to formulate unambiguous, complete statements because they have a clearly defined syntax and semantics. In general, a programming language guarantees that all syntactically correct expressions can be transformed automatically in machine-executable code, while this must not be true for a specification language. Specification languages and high-level declarative programming languages (i. e., functional and logic languages) share the common characteristic, that they abstract from the "how to solve" a problem on a machine and focus on the "what to solve", that is, they have more expressive power and provide much abstracter constructs than typical imperative programming languages (like *C*). Nevertheless, generating specifications or declarative programs requires experts who are trained in representing problems correctly and completely – that is, giving the desired input/output relations and covering all possible inputs.

For that reason, inductive program synthesis addresses the problem to derive programs from *incomplete specifications*. The basic idea is, that a user presents some examples of the desired program behavior. Specification by examples is incomplete, because the synthesis algorithm must generalize the desired program behavior from *some* inputs to *all* possible inputs. Furthermore, the examples themselves can contain more or less information. Examples might be presented simply as *input/output pairs*. For structural list-problems (as *last* or *reverse*), generic input/output pairs, abstracting from concrete values of list elements can be given. This makes the examples less ambiguous.

More information about the desired program can be provided, if examples are represented as *traces* which illustrate the operation of a program. Traces can prescribe the to be used data structures and operations, if the inputs are described by tests and outputs by transformations of the inputs using predefined operators. Traces are called complete, if all information about data structures changed, operators applied, and control decisions taken is given. Additionally,

examples are more informative, if they indicate what kind of computations are *not* desired in the goal program. This can be done by presenting positive and negative examples. Another strategy is to present examples for the first $k$ inputs, thereby defining an order over the input domain. Of course, the more information is presented by an example specification, the more the system user has to know about the structure of the desired program.

The kind and number of examples must suffice to specify what the desired program is supposed to calculate. Sufficiency is dependent on the synthesis algorithm which is applied. In general, it is necessary to present more than one example, because otherwise a program with constant output is the most parsimonious induction.

### 1.2.2 SYNTHESIS METHODS

Deductive and inductive synthesis are special cases of deductive and inductive inference. Therefore, we first give a general characterization of deduction and induction:

**Deductive versus Inductive Inference.** Deductive inference addresses the problem of inferring new facts or rules (called theorems) from a set of given facts and rules (called theory or axioms) which are assumed to be true. For example, from the axioms

1. *list(l)* $\rightarrow \neg$ *null(cons(x, l))*
2. *list([A,B,C])*

we can infer theorem

3. $\neg$ *null(cons(Z, [A,B,C])).*

Deductive approaches are typically based on a logical calculus. That is, axioms are represented in a formal language (such as first order predicate calculus) and inference is based on a syntactical proof mechanism (such as resolution). An example for a logical calculus was given in section 2.2 in chapter 2 (situation calculus). The theorem given above can be proved by resolution in the following way:

1. *null(cons(Z, [A,B,C]))* (Negation of the theorem)
2. $\neg$ *list(l)* $\vee \neg$ *null(cons(x, l))* (axiom 1)
3. $\neg$ *list([A,B,C])* (Resolve 1, 2 with substitutions $\sigma = \{x/Z, l/[A, B, C]\}$)
4. *list([A,B,C])* (axiom 2)
5. *contradiction* (Resolve 3, 4).

Inductive inference addresses the problem of inferring a generalized rule which holds for a domain (called hypothesis) from a set of observed instances belonging to this domain (called examples). For example, from the examples

- *list([Z,A,B,C])*
- *list([1,2,3])*
- *list([R,B])*

axiom (1) given above might be inferred as hypothesis for the domain of (flat) lists. For this inference, it is necessary to refer to some background knowledge about lists, namely that lists are constructed by means of the list-constructor *cons(x, l)* and that *null(l)* is true if $l$ is the empty list and false otherwise. Other hypotheses which could be inferred are

- *list(l)*

which is an over-generalization because it includes objects which are no lists (i. e., atoms) or

- $l \in \{[Z, A, B, C], [1, 2, 3], [R, B]\} \rightarrow list(l)$

which is no generalization but just a compact representation of the examples.

Inductive approaches are realized within the different frameworks offered by machine learning research. The language for representing hypotheses depends on the selected framework. Examples are decision trees, a subset of predicate calculus, or functional programs. Also depending on the framework, hypothesis construction can be constrained more or less by the presented examples. In extreme, hypotheses might be generated just by enumeration (Gold, 1967): a legal expression of the hypothesis language is generated and tested against the examples. If the hypothesis is not consistent with the examples, it is rejected and a new hypothesis is generated. Note, that testing whether a hypothesis holds for an example corresponds to deductive inference (Mitchell, 1997, p. 291). An overview of machine learning is given by Mitchell (1997) and we will present approaches to inductive inference in section 3.

Deduction guarantees that the inference is correct (with respect to the given axioms), while induction only results in a hypothesis. From the perspective of epistemology, one might say that a deductive proof does not generate knowledge – it explicates information which is already contained in the given axioms. Inductive inference results in new information, but the inference has only hypothetical status which holds as long as the system is not confronted with examples which contradict the inference.

To make this difference more explicit, lets look at a set-theoretic interpretation of inference:

**Definition 6.1 (Relations in a Domain)** *Let $\mathcal{D}$ be a set of objects belonging to a domain. A relation with arity $i$ is given as $R^i \subseteq \mathcal{D}^i$. The set of all relations which hold in a domain is $\mathcal{R}$.*

For a given domain, for example the set of flat lists, the set of all axioms which hold in this domain is extensionally given by relations. For example,

there might be an unary relation $R$ containing all lists, and a binary relation $R'$ containing all pairs of lists *(l, cons(x, l))* where $l \in R$. In general, relations can represent formulas of arbitrary complexity.

Now we can define (the semantics of) deduction and induction in the following way:

**Definition 6.2 (Deduction)** *Given a set of axioms $\mathcal{A} \subset \mathcal{R}$, deductive inference means to* decide *whether for a theorem $R \notin \mathcal{A}$ holds $R \in \mathcal{R}$.*

In a deductive (proof) calculus, $R \in \mathcal{R}$ is decided by showing that $R$ can be derived from the given axioms $\mathcal{A}$, that is, $\mathcal{A} \models R$. A standard *syntactical* approach to deductive inference is for example resolution (see illustration above).

**Definition 6.3 (Induction)** *Given a set of examples $\mathcal{E} \subset \mathcal{R}$, inductive inference means to* search *a hypothesis $\mathcal{H} = \{R_1, \ldots R_n\}$ such that $\mathcal{H} \neq \mathcal{E}$ and $\forall\, E_i \in \mathcal{E} : E_i \in \mathcal{H}$.*

The search for a hypothesis takes place in the set of possible hypotheses given a fixed hypotheses language, for example, the set of all syntactically correct Lisp programs. Typically, selection of a hypothesis is not only restricted by demanding that $\mathcal{H}$ explains ("covers") all presented examples, but by additional criteria such as simplicity.

**Deductive versus Inductive Synthesis.** In program synthesis, the result of inference is a computer program, which transforms all legal inputs $x$ in the desired output $y = f(x)$. For deductive synthesis, the complete formal specification of the pre- and post-conditions of the desired program can be represented as theorem of the following form:

**Definition 6.4 (Program Specification as Theorem)**

$$\forall\, x\, \exists\, y\, [Pre(x) \Rightarrow Post(x, y)].$$

For example, for the specification of the *last*-program (see tab. 6.1), *Pre(x)* states that $x$ is a non-empty list and *Post(x, y)* states, that $y$ is the last element of list $x$. A constructive theorem prover (such as Green's approach, described in sect. 2.2 in chap. 2) tries to prove that this theorem holds. If the proof fails, there exists no feasible program (given the axioms on which the proof is based); if the proof succeeds, a program $f(x)$ is returned as result of the constructive proof. For $f(x)$ must hold

**Definition 6.5 (Program Specification as Constructive Theorem)**

$$\forall\, x\, [Pre(x) \Rightarrow Post(x, f(x))]$$

In the proof the existential quantified variable $y$ must be explicitly constructed as a function over $x$ (Skolemization).

An alternative to theorem proving is *program transformation*. In theorem proving, each proof step consists of rewriting the current program (formula) by selecting a given axiom and applying the proof method (e. g., resolution). In program transformation, rewriting is based on transformation rules.

**Definition 6.6 (Transformation Rule)** *A transformation rule is defined as a conditioned rule:*

**If** Application-Condition **Then** Left-Hand-Pattern $\rightarrow$ Right-Hand-Pattern.

The rules might be given in equational logic (i. e., a special class of axioms) or they might be uni-directional. Each transformation step consists of rewriting the current program by replacing some part which matches with the *Left-Hand-Pattern* by another expression which matches with the *Right-Hand-Pattern*.

In section 2 we will present theorem proving as well as transformational approaches to deductive synthesis.

For inductive synthesis, a program is constructed as generalization over the given incomplete specification (input/output examples or traces, see tab. 6.1) such that the following proposition holds:

**Definition 6.7 (Characteristic of an Induced Program)**

$$\forall\ (x, y)\ \in \mathcal{E}[f(x) = y]$$

*where $\mathcal{E}$ is a set of examples with $x$ as possible input in the desired program and $y$ as corresponding output value or trace for $x$.*

The formal foundation of inductive synthesis is *grammar inference*. Here the examples are viewed as words which belong to some unknown formal language and the inference task is to construct a grammar (or automaton) which generates (or recognizes) a language to which the example words belong. The classical approach to program synthesis is the construction of recursive functional (mostly Lisp) programs from traces. Such traces are either given as input our automatically constructed from input/output examples. Alternatively, in the context of inductive logic programming, the construction of logical (Prolog) programs from input/output examples is investigated. In both research areas, there are methods which base hypothesis construction strongly on the presented examples versus methods which depend more on search in hypothesis space (i. e., generate and test). An approach which is based explicitly on generate and test is genetic programming.

In section 3 we will present grammar inference, functional program synthesis, inductive logic programming, and genetic programming as approaches to inductive program synthesis.

**Enriching Program Synthesis with Knowledge.** The basic approaches to deductive and inductive program synthesis depend on a limited amount of knowledge: a set of (correct) axioms or transformation rules in deduction; or a restricted hypothesis language and some rudimentary background knowledge (for example about data types and primitive operators) in induction. These basic approaches can be enriched by additional knowledge. Such knowledge might be not universally valid but can make a system more powerful. As a consequence, a basic fully automatic, algorithmic approach is extended to an interactive, heuristic approach.

A successful approach to knowledge-based program synthesis is to guide synthesis by pre-defined program schemes, also called *design strategies*. A program scheme is a program template (for example represented in higher order logic) representing a fixed over-all program structure (i. e., a fixed flow of control) but abstracting from concrete operations. For a new programming problem, an adequate scheme is selected (by the user) from the library and a program is constructed by stepwise refinement of this scheme. For example, a *quicksort* program might be synthesized by refining a *divide-and-conquer* scheme. Scheme-based synthesis is typically realized within the deductive framework (Smith, 1990). An approach to combine inductive program synthesis and schema-refinement is proposed by Flener (1995).

A special approach to inductive program synthesis is programming by analogy, that is, *program reuse*. Here, already known programs (predefined or previously synthesized) are stored in a library. For a new programming problem, a similar problem and the program which solves this problem are retrieved and the new program is constructed by modifying the retrieved program. Analogy can be seen as a special case of induction because modification is based on the structural similarity of both problems and structural similarity is typically obtained by generalizing over the common structure of the problems (see calculation of least general generalizations in section 3.3 and anti-unification in chapter 7 and chapter 12).

We will come back to schema-based synthesis and programming by analogy in Part III, addressing the problems of (1) automatic acquisition of abstract schemes from example problems, and (2) automatical retrieval of an appropriate schema for a given programming problem.

## 1.3 POINTERS TO LITERATURE

Although program synthesis is an active area of research since the beginning of computer science, program synthesis is not covered in text books on programming, software engineering, or AI. The main reason for that omission might be that program synthesis research does not rely on one common formalism but that each research group proposes its own approach. This is even the case within a given framework – as synthesis by theorem proving

or inductive logic programming. Furthermore, the formalisms are typically quite complex, such that it is difficult to present a formalism together with a complete, non-trivial example in a compact way.

An introductory overview to knowledge-based software engineering is given by Lowry and Duran (1989). Here approaches to and systems for specification acquisition, specification validation and maintenance, as well as to deductive program synthesis are presented. A collection of influential papers in AI and software engineering was edited by Rich and Waters (1986b). The papers cover a large variety of research areas including deductive and inductive program synthesis, program verification, natural language specification, knowledge based approaches, and programming tutors. Collections of research papers on AI and software engineering are presented by Lowry and McCarthy (1991), Partridge (1991), and Messing and Campbell (1999).

An introductory overview to program synthesis is given by (Barr and Feigenbaum, 1982). Here the classical approaches to and systems for deductive and inductive program synthesis of functional programs are presented. A collection of influential papers in program synthesis was edited by Biermann, Guiho, and Kodratoff (1984), addressing deductive synthesis, inductive synthesis of functional programs, and grammar inference. A more recent survey of program synthesis is given by Flener (1995, part 1) and a survey of inductive logic program synthesis is given by Flener and Yilmaz (1999).

Current research in program synthesis is for example covered in the journal *Automated Software Engineering*. AI conferences, especially machine learning conferences (ECML, ICML) typically have sections on inductive program synthesis.

## 2 DEDUCTIVE APPROACHES

In the following we give a short overview of approaches to deductive program synthesis. First we introduce constructive theorem proving and afterwards program transformation.

### 2.1 CONSTRUCTIVE THEOREM PROVING

Automated theorem proving has in general not to be constructive. That means, for example, that from a statement $\neg\forall x \neg P(x)$ can be followed that $\exists x P(x)$ without the necessity to actually construct an object $a$ which has property $P$. In contrast, for a proof to be constructive, such an object $a$ has to be given together with a proof that $a$ has property $P$ (Thompson, 1991). Classical theorem provers, which are applied for program *verification*, such as the Boyer-Moore theorem prover (Boyer and Moore, 1975) cannot be used for program *synthesis* because they cannot reason constructively about existential quantifications. As introduced above, a constructive proof of the statement

$\forall x \exists y Pre(x) \rightarrow Post(x, y)$ means that a program $f$ must be constructed such that for all inputs $a$, for which $Pre(a)$ holds, $Post(x, f(x))$ holds.

The observation that constructive proofs correspond to programs was made by a lot of different researchers (e. g., Bates and Constable, 1985) and was explicitly stated as so-called Curry-Howard isomorphism in the eighties. One of the oldest approaches to constructive theorem proving was proposed by Green (1969), introducing a constructive variant of resolution. In the following, we first present this pioneering work, afterwards we describe the deductive tableau method of (Manna and Waldinger, 1980), and give a short survey of more contemporary approaches.

### 2.1.1 PROGRAM SYNTHESIS BY RESOLUTION

Green (1969) introduced a constructive version of clausal resolution and demonstrated with his system QA3 that constructive theorem proving can be applied for constructing plans (see sect. 2.2 in chap. 2) and for automatic (Lisp) programming. For automatic programming, the theorem prover needs two sets of axioms: (1) Axioms defining the functions and constructs of (a subset of) a programming language (such as Lisp), and (2) axioms defining an input/output relation *R(x, y)*[1], which is true if and only if $x$ is an appropriate input for some program and $y$ is the corresponding output generated by this program. Green addresses four types of automatic programming problems:

**Checking:** Proving that *R(a,b)* is true or false.
For a fixed input/output pair it is checked whether a relation *R(a, b)* holds.

**Simulation:** Proving that $\exists x\ R(a, x)$ is true (returning $x = b$) or false.
For a fixed input value $a$, output $b$ is constructed.

**Verification:** Proving that $\forall x\ R(x, f(x))$ is true or false (returning $x = c$ as counter-example).
For a user-constructed program $f(x)$, its correctness is checked. For proofs concerning looping or recursive programs, induction axioms are required to proof convergence (termination).

**Synthesis:** Proving that $\forall x\ \exists y\ R(x, y)$ is true (returning the program $y = f(x)$) or false (returning $x = c$ as counter-example).
Induction axioms are required to synthesize recursive programs.

Green gives two examples for program synthesis: the synthesis of a simple non-recursive program which sorts two numbers of a *dotted pair*, and the synthesis of a recursive function for sorting.

[1] Relation *R(x,y)* corresponds to relation *Post(x,y)* given in definition 6.4.

**Synthesis of a Non-Recursive Function.** For the dotted pair problem, the following axioms concerning Lisp are given:

1. *x = car(cons(x,y))*
2. *y = cdr(cons(x,y))*
3. *x = nil* $\rightarrow$ *cond(x,y,z) = z*
4. *x* $\neq$ *nil* $\rightarrow$ *cond(x,y,z) = y*
5. $\forall x, y$ *[lessp(x,y)* $\neq$ *nil* $\leftrightarrow x < y$*].*

The axiom specifying the desired input/output relation is given as

$$\forall x \exists y \quad [car(x) < cdr(x) \rightarrow y = x] \wedge \\ [(car(x) \geq cdr(x)) \rightarrow (car(x) = cdr(y) \wedge cdr(x) = car(y))].$$

The axioms states that a dotted pair $(x_1.x_2)$ which is already sorted ($x_1 < x_2$) is just returned, otherwise, the reversed pair must be returned. By resolving the input/output axioms with axiom (5), the mathematical expressions ($<$, $\leq$) are replaced by Lisp-expressions $lessp$. Resolving expressions *lessp(x,y)* $=$ *nil* with axioms (3) and (4) introduces conditional expressions; and axioms (1) and (2) are used to introduce *cons*-expressions. The resulting program is

*y = cond(lessp(car(x),cdr(x)), x, cons(cdr(x),car(x))).*

**Synthesis of a Recursive Function.** To synthesize a recursive program the theorem prover additionally requires an *induction axiom*. For example, for a recursion over finite linear lists, it can be stated that the empty list is reached in a finite number of steps by stepwise applying *cdr(l)*:

**Definition 6.8 (Induction over Linear Lists)**

$$[P(h(nil)) \wedge \forall x\, [\neg atom(x) \wedge P(h(cdr(x))) \rightarrow P(h(x))]] \rightarrow \forall z\, P(h(z))$$

*where $P$ is a predicate and $h$ is a function.*

The axiom specifying the desired input/output relation for a list-sorting function can be stated for example as:

$$\forall x\, \exists y\, [R(nil, y) \wedge [[\neg atom(x) \wedge R(cdr(x), sort(cdr(x)))] \rightarrow R(x, y)]].$$

The searched for function is already named as *sort*.

Given the Lisp axioms above, some additional axioms characterizing the predicates *atom(x)* and *equal(x,y)*, and axioms specifying a pre-defined function *merge(x,l)* (inserting number $x$ in a sorted list $l$ such that the result is a sorted list), the following function can be synthesized:

*y = cond(equal(x,nil), nil, merge(car(x),sort(cdr(x)))).*

**Drawbacks of Theorem-Proving.** For a given set of axioms and a complete, formal specification, program synthesis by theorem-proving results in a program which is correct with respect to the specification. The resulting program is constructed as side-effect of the proof by instantiation of variable $y$ in the input/output relation $R(x, y)$.

Fully automated theorem proving has several disadvantages as approach to program synthesis:

- The domain must be axiomatized completely.
  The given axiomatization has a strong influence on the complexity of search for the proof and it determines the way in which the searched for program is expressed. For example, Green (1969) reports that his theorem prover could not synthesize *sort* in a "reasonable amount of time" with the given axiomatization. Given a different set of axioms (not reported in the paper), QA3 created the program *cond(x, merge(car(x),sort(cdr(x))), nil)*.
  Providing a set of axioms which are sufficient to prove the input/output relation and thereby to synthesize a program, presupposes that a lot is already known about the program. For example, to synthesize the *sort* program, an axiomatization of the *merge* function had to be provided.

- It can be more difficult to give a correct and complete input/output specification than to directly write the program.
  For the *sort* example given above, the searched for recursive structure is already given in the specification by separating the case of an empty list as input from the case of a non-empty list and by presenting the implication from the rest of a list (*cdr(x)*) to the list itself.

- Theorem provers lack the power to produce proofs for more complicated specifications.
  The main source of inefficiency is that a variety of induction axioms must be specified for synthesizing recursive programs. The selection of the "suitable" induction scheme for a given problem is crucial for finding a solution in reasonable time. Therefore, selection of the induction scheme to be used is often performed by user-interaction!

This disadvantages of theorem proving are true for the original approach of Green and are still true for the more sophisticated approaches of today, which are not restricted to proof by resolution but use a variety of proof mechanisms (see below). The originally given hard problem of automatic program construction is reformulated as equally hard problem of automatic theorem proving (Rich and Waters, 1986a). Nevertheless, the proof-as-program approach is an important contribution to program synthesis: First, the necessity of complete and formal specifications compels the program designer to state all knowledge involved in program construction explicitly. Second, and more important, while theorem

proving might not be useful in isolation, it can be helpful or even necessary as part of a larger system. For example, in transformation systems (see sect. 2.2), the applicability of rules can be proved by showing that a precondition holds for a given expression.

### 2.1.2 PLANNING AND PROGRAM SYNTHESIS WITH DEDUCTIVE TABLEAUS

A second influential contribution to program synthesis as constructive proof was made by Manna and Waldinger (1980). As Green, they address not only deductive program synthesis but also deductive plan construction (Manna and Waldinger, 1987).

**Deductive Tableaus.** The approach of Manna and Waldinger is also based on resolution. In contrast to the classical resolution (Robinson, 1965) used by Green, their resolution rule does not depend on axioms represented in clausal form. Their proposed formalism – the deductive tableaus – incorporates not only non-clausal resolution but also transformation rules and structural induction. The transformation rules have been taken over from an earlier, transformation based programming system called *Dedalus* (Manna and Waldinger, 1975, 1979).

A deductive tableau consists of three columns

| Assertions | Goals | Outputs |
|---|---|---|
| *Pre(x)* | | |
| | *Post(x,y)* | *y* |

with $x$ as input variable, $y$ as output variable, *Pre(x)* as precondition and *Post(x,y)* as postcondition of the searched for program. The initial tableau is constructed from a specification of the form (see also table 6.1)

$f(x) \Leftarrow$ find $y$ such that $Post(x, y)$ where $Pre(x)$.

In general, the semantics of a tableau with assertions $A_i(x)$ and goals $G_j(x, y)$ is

$$\forall x A_1(x) \wedge \ldots \wedge A_n(x) \rightarrow \exists y G_1(x, y) \vee \ldots \vee G_n(x, y).$$

A proof is performed by adding new rows to the tableau through rules of logical inference. A proof is successful if a row could be constructed where the goals column contains the expression “true” and the output column contains a term which consists only of primitive expressions of the target programming language.

As an example for a derivation step, we present a simplified version of the so called GG-resolution (omitting possible unification of sub-expressions): For

two goals – $F$ in column $i$ and $G$ in column $j$ – a new row can be constructed with the goal entry

$$F[P \leftarrow true] \wedge G[P \leftarrow false]$$

where $P$ is a common sub-expression of $F$ and $G$. For the output column, GG-resolution results in the introduction of a conditional expression:

| Assertions | Goals | Outputs |
|---|---|---|
| | $a \geq b$ | a |
| | $\neg(a \geq b)$ | b |
| | *true* $\wedge \neg$ *false* | if $a \geq b$ then $a$ else $b$ |
| | true. | |

Again, induction is used for recursion-formation:

**Definition 6.9 (Induction Rule)** *For a searched for program $f(x)$ with assertion $Pre(x)$ and goal $Post(x, y)$ the induction hypothesis is $\forall u(u \prec x) \rightarrow (Pre(u) \rightarrow Post(u, f(u)))$ where $\prec$ is a well-founded ordering over the set of input data $\{x\}$.*

**Recursive Plans.** Originally, Manna and Waldinger applied the deductive tableau method to synthesize functional programs, such as reversing lists or finding the quotient of two integers (Manna and Waldinger, 1992). Additionally, they demonstrated how imperative recursive programs can be synthesized by deductive planning (Manna and Waldinger, 1987). As example they used the problem of clearing a block (see also sect. 1.4.1 in chap. 3).

The searched for plan is called *makeclear(a)* where $a$ denotes a block in a blocks-world. Plan construction is based on proving the theorem

$$\forall s_0 \forall a \exists z_1 [Clear(s_0; z_1, a)]$$

meaning "for an initial state $s_0$, the block $a$ is clear after execution of plan $z_1$".

Among the pre-specified axioms for this problems are:

**hat-axiom:** If ¬ Clear(s, x) Then On(s, hat(s,x), x)
*(If block $x$ is not clear in situation $s$ then a block $hat(s, x)$ is lying on block $x$ in situation $s$ where $hat(s, x)$ is the block which lies on $x$ in $s$.)*

**put-table-axiom:** If Clear(s, x) Then On(put(s, x, table), x, table)
*(If block $x$ is clear in situation $s$ then it lies on the table in a situation where $x$ was put on the table immediately before.)*

and the resulting plan (program) is

*makeclear(a)* $\Leftarrow$
  *If Clear(a)*
  *Then* $\Lambda$ (the current situation)
  *Else makeclear(hat(a));put(hat(a),table)*.

The complete derivation of this plan using deductive tableaus is reported in Manna and Waldinger (1987).

### 2.1.3 FURTHER APPROACHES

A third "classical" approach to program synthesis by theorem proving was proposed by (Bibel, 1980). Current automatic theorem provers, such as *Nuprl*[2] or *Isabelle*[3] can be in principle used for program synthesis. But up to now, pure theorem proving approaches can only be applied to small problems. Current research addresses the problem of enriching constructive theorem proving with knowledge-based methods – such as proof tactics or program schemes (Kreitz, 1998; Bibel, Korn, Kreitz, and Kurucz, 1998).

## 2.2 PROGRAM TRANSFORMATION

Program transformation is the predominant technique in deductive program synthesis. Typically, directed rules are used to transform an expression into a syntactically different, but semantically equivalent expression. There are two principal kinds of transformation: *lateral* and *vertical* transformation (Rich and Waters, 1986a). Lateral transformation generates expressions on the same level of abstraction. For example, a linear recursive program might be rewritten into a more efficient tail-recursive program. Program synthesis is realized by vertical transformation – rewriting a specification into a program by applying transformation rules which represent the relationship between constructs on an abstract level and constructs on program level. If the starting point for transformation is already an executable (high-level) program, one speaks of *transformational implementation*.

In the following, we first present the pioneering approach of Burstall and Darlington (1977). The authors present a small set of powerful rules for transforming given programs in more efficient programs, that is they propose an approach to transformational implementation. The main contribution of their approach is the introduction of rules for unfolding and folding (synthesizing) recursive programs. Afterwards, we present the CIP (computer-aided intuition-guided programming) approach (Broy and Pepper, 1981) which focuses on correctness-preserving transformations. CIP does not aim at full automatization, it can be classified as a *meta-programming* approach (Feather,

[2] see `http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`
[3] see `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`

1987) which supports a program-developer in the construction of correct programs. Finally, we present the KIDS-system (Smith, 1990) which is the most successful deductive synthesis system today. KIDS is a program synthesis system which transforms initial, not necessarily executable, specifications into efficient programs by *stepwise refinement*.

### 2.2.1 TRANSFORMATIONAL IMPLEMENTATION: THE FOLD UNFOLD MECHANISM

*The mind is a cauldron and things bubble up and show for a moment, then slip back into the brew. You can't reach down and find anything by touch. You wait for some order, some relationship in the order in which they appear. Then yell Eureka! and believe that it was a process of cold, pure logic.*

—Travis McGee in: John D. Mac Donald, *The Girl in the Plain Brown Wrapper*, 1968

Starting point for the transformation approach of Burstall and Darlington (1977) is a program, which is presented as a set of equations with left-hand sides representing program heads and right-hand sides representing calculations.

For example, the Fibonacci function is presented as

1. fib(0) $\Leftarrow$ 1
2. fib(1) $\Leftarrow$ 1
3. fib(x+2) $\Leftarrow$ fib(x+1) + fib(x).

The following inference rules for transforming recursive equations are given:

**Definition:** Introduce a new recursive equation whose left-hand expression is not an instance of the left-hand expression of any previous equation.
We will see below, that introduction of a suitable equation in general cannot be performed automatically ("eureka" step).

**Instantiation:** Introduce a substitution instance of an existing equation.

**Unfolding:** If $E \Leftarrow E'$ and $F \Leftarrow F'$ are equations and there is some occurrence in $F'$ of an instance of $E$, replace it by the corresponding instance of $E'$, obtaining $F''$ and add equation $F \Leftarrow F''$.
This rule corresponds to the expansion of a recursive function by replacing the function call by the function body with according substitution of the parameters.

**Folding:** If $E \Leftarrow E'$ and $F \Leftarrow F'$ are equations and there is some occurrence in $F'$ of an instance of $E'$, replace it by the corresponding instance of $E$, obtaining $F''$ and add equation $F \Leftarrow F''$.
This is the "inverse" rule to unfolding. We will discuss folding of terms in detail in chapter 7.

**Abstraction:** Introduce a **where** clause by deriving from a previous equation $E \Leftarrow E'$ a new equation

$$E \Leftarrow E'[u_1/F_1, \ldots, u_n/F_n] \; where \; \langle u_1, \ldots, u_n \rangle = \langle F_1, \ldots, F_n \rangle.$$

**Laws:** Algebraic laws – such as associativity or commutativity – can be used to rewrite the right-hand sides of equations.

The authors propose the following *strategy* for application of rules

- Make any necessary definitions.
- Instantiate.
- For each instantiation unfold repeatedly. At each stage of unfolding:
  - Try to apply laws and abstractions.
  - Fold repeatedly.

**Transforming Recursive Functions.** We will illustrate the approach, demonstrating how the *fibonacci* program given above can be transformed into a more efficient program which avoids calculating values twice:*transformation, recursive function*

1. fib(0) $\Leftarrow$ 1 (given)
2. fib(1) $\Leftarrow$ 1 (given)
3. fib(x+2) $\Leftarrow$ fib(x+1) + fib(x) (given)
4. g(x) $\Leftarrow$ $\langle$fib(x+1),fib(x) $\rangle$ (definition, eureka!)
5. g(0) $\Leftarrow$ $\langle$fib(1),fib(0) $\rangle$ (instantiation)
   g(0) $\Leftarrow$ $\langle$1,1 $\rangle$ (unfolding with 1, 2)
6. g(x+1) $\Leftarrow$ $\langle$fib(x+2),fib(x+1) $\rangle$ (instantiate 4)
   g(x+1) $\Leftarrow$ $\langle$fib(x+1)+fib(x),fib(x+1) $\rangle$ (unfold with 3)
   g(x+1) $\Leftarrow$ $\langle$u+v,u$\rangle$ where $\langle$u,v$\rangle$ = $\langle$fib(x+1),fib(x) $\rangle$ (abstract)
   g(x+1) $\Leftarrow$ $\langle$u+v,u$\rangle$ where $\langle$u,v$\rangle$ = g(x) (fold with 4)
7. fib(x+2) $\Leftarrow$ u + v where $\langle$u,v$\rangle$ = $\langle$fib(x+1),fib(x) $\rangle$ (abstract 3)
   f(x+2) $\Leftarrow$ u + v where $\langle$u,v$\rangle$ = g(x) (fold with 4)

The new, more efficient definition of *fibonacci* is:

fib(0) $\Leftarrow$ 1

fib(1) $\Leftarrow$ 1

fib(x+2) ⇐ u + v where ⟨u,v⟩ = g(x)

g(0) ⇐ ⟨1,1⟩

g(x+1) ⇐ u + v where ⟨u,v⟩ = g(x).

**Abstract Programming and Data Type Change.** Burstall and Darlington (1977) also demonstrated how vertical program transformation can be realized by transforming abstract programs, defined on high-level, abstract data types, into concrete programs defined on concrete data.

Again, we illustrate the approach with an example. Given is the abstract data type of labeled trees:

**Abstract Data Type**

$$niltree \in labeledTrees$$

$$ltree : atoms \times labeledTrees \times labeledTrees \rightarrow labeledTrees.$$

That is, a labeled tree is either empty (*niltree*) or it is a labeled node (*atoms*) which branches into two labeled trees.

Furthermore, a data type of binary trees might be available as basic data structure (e. g., in Lisp) with constructors *nil* and *pair*:

**Concrete Data Structure**

$$nil \in binaryTrees$$

$$atoms \in binaryTrees$$

$$pair : binaryTrees \times binaryTrees \rightarrow binaryTrees.$$

A labeled tree can be represented as binary tree by representing each node as a pair of a label and a binary tree. For example:

*ltree(A, niltree, ltree(B, niltree, niltree))*

can be represented as

*pair(A, pair(nil, pair(B, pair(nil, nil))))*.

The relationship between the abstract data type and the concrete data structure can be expressed by the following representation function:

$$R : binaryTrees \rightarrow labeledTrees$$

$$R(nil) \Leftarrow niltree$$

$$R(pair(a, (pair(p_1, p_2)))) \Leftarrow ltree(a, R(p_1), R(p_2)).$$

The inverse representation function – which in some cases could be generated automatically – defines how to *code* the data type:

$$C : labeledTrees \rightarrow binaryTrees$$

$$C(niltree) \Leftarrow nil$$

$$C(ltree(a, t_1, t_2)) \Leftarrow pair(a, pair(C(t_1), C(t_2))).$$

The user can write an abstract program, using the abstract data type. Typically, writing abstract programs involves less work than writing concrete programs because implementation specific details are omitted. Furthermore, abstract programs are more flexible because they can be transformed into different concrete realizations.

An example for an abstract program is *twist* which mirrors the input tree:

$$twist : labeledTrees \rightarrow labeledTrees$$

$$twist(niltree) \Leftarrow niltree$$

$$twist(ltree(a, t_1, t_2)) \Leftarrow ltree(a, twist(t_2), twist(t_1))$$

The goal is, to automatically generate a concrete program *TWIST(p) = C(twist(R(p)))*:

1. TWIST(p) ⇐ C(twist(R(p)))
2. TWIST(nil) ⇐ C(twist(R(nil))) (instantiate)
3. **TWIST(nil) ⇐ nil** (unfold C, twist, R, and evaluate)
4. TWIST(pair(a,pair($p_1$,$p_2$))) ⇐ C(twist(R(pair(a,pair($p_1$,$p_2$))))) (instantiate)
5. TWIST(pair(a,pair($p_1$,$p_2$))) ⇐ C(twist(ltree(a,R($p_1$), R($p_2$)))) (unfold R)
6. TWIST(pair(a,pair($p_1$,$p_2$))) ⇐ C(ltree(a,twist(R($p_2$)),twist(R($p_1$)))) (unfold twist)
7. TWIST(pair(a,pair($p_1$,$p_2$))) ⇐ pair(a,pair(C(twist(R($p_2$))), C(twist(R($p_1$))))) (unfold C)
8. **TWIST(pair(a,pair($p_1$,$p_2$))) ⇐ pair(a,pair(TWIST($p_2$), TWIST($p_1$)))** (fold with 1)

with the resulting program given by equations 3 and 8.

**Characteristics of Transformational Implementation.** To sum up, the approach proposed by Burstall and Darlington (1977) has the following characteristics:

**Specification:** Input is a structurally simple program whose correctness is obvious (or can easily be proved).

**Basic Rules:** A transformation system is based on a set of rules. There are rules which define semantics-preserving re-formulations of the (right-hand side, i. e., body) program (or specification). Other rules, such as definition and instantiation, cannot be applied mechanical but are based on creativity (eureka) and insight into the problem to be solved.

**Partial Correctness:** Transformations are semantics-preserving modulo termination.

**Selection of Rules:** Providing that sequence of rule application which leads to the desired result typically cannot be performed fully automatically. Burstall and Darlington presented a simple strategy for rule selection. This strategy can in general not be applied without guidance by the user.

### 2.2.2 META-PROGRAMMING: THE CIP APPROACH

The CIP approach was developed during the seventies and eighties (Broy and Pepper, 1981; Bauer, Broy, Möller, Pepper, Wirsing, et al., 1985; Bauer, Ehler, Horsch, Möller, Partsch, Paukner, and Pepper, 1987) as a system to support the formal development of correct and efficient programs (meta-programming). The approach has the following characteristics:

**Transformation Rules:** Describe mappings from programs to programs. They are represented as directed pairs of program schemes ($a \rightarrow b$ together with an enabling condition ($c$) defining applicability.

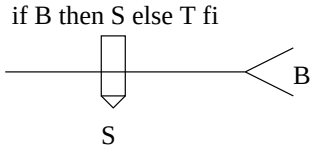
**Transformational Semantics:** A programming language is considered as term algebra. The given transformation rules define congruence relations on this algebra and thereby establish a notion of "equivalence of programs" (Ehrig and Mahr, 1985).

**Correctness of Transformations:** The correctness of a basic set of transformation rules can be proved in the usual way – showing that $a$ and $b$ are equivalent with respect to the properties of the semantic model. Correctness of all other rules can be shown by *transformational proofs*: A transformation rule $T : a \rightarrow b$ is correct if it can be deduced from a set of already verified rules $T_1, \ldots T_n$: $a \overset{T_1}{\mapsto} a_1 \overset{T_2}{\mapsto} \ldots \overset{T_n}{\mapsto} a_n = b$. In general, for all rules $a \rightarrow b$, it must hold, that the relation between $a$ and $b$ is reflexive and transitive. Properties of recursive programs are proved by *transformational induction*:

For two recursive programs $p$ and $q$ with bodies $\tau[p]$ and $\sigma[q]$ every call $p(x)$ can be transformed into a call $q(x)$ if there is a transformation rule $\tau[y] \rightarrow \sigma[y]$.

**Abstract Data Types:** Abstract data types are given by a signature (sorts and operations) and a set of equations (axioms) (see, e. g. Ehrig and Mahr, 1985). These axioms can be considered as transformation rules which are applicable within the whole scope of the type. (Considering a programming language as term algebra, the programming language itself can be seen as abstract data type with the transformation rules as axioms).

**Assertions:** For some transformation rules there might be enabling conditions which hold only locally. For example, for the rule




the enabling condition $B$ can be given or not, depending on the current input. If $B$ holds, than the conditional statement evaluates to $S$. Such local properties are expressed by assertions which can be provided by the programmer. Furthermore, transformation rules can be defined for generating or propagating assertions.

Again, we illustrate the approach with an example, which is presented in detail in (Broy and Pepper, 1981). The searched for program is a realization of the Warshall-algorithm for calculating the transitive closure of a graph.

The graph is given by its characteristic function

- **function** *edge (x:node, y:node):bool;*
  (*true*, if nodes $x$ and $y$ are directly connected).

A path is defined as sequences of nodes where each pair of succeeding nodes are directly connected:

- **function** *ispath (s:seq(node)):bool;*
  $\forall s_1, s_2$*: seq(node), x,y: node::*
  $s = s_1 \circ x \circ y \circ s_2 \Rightarrow edge(x, y)$.

Starting point for program transformation is the specification of the searched for function:

- **function** *trans (x:node, y:node):bool;*
  $\exists$ *P: seq(node):: ispath(*$x \circ p \circ y$*).*

The basic idea (eureka!) is to define a more general function, which only considers the first $i$ nodes (where nodes are given as natural numbers $1 \ldots n$):

- **function** *tc(i:nat, x:node, y:node):bool;*
  $\exists$ *P: seq(node):: ispath(*$x \circ p \circ y$*)*
  $\wedge \forall$ *z: node::* $z \in p \Rightarrow z \leq i$.

In a first step, it must be proved, that *tc(n,x,y)* $=$ *trans(x,y)*, where $n$ is the number of all nodes in a graph:

- *Lemma 1:* $tc(n, x, y) = trans(x, y)$
- *tc(n,x,y)* = $\exists$ *P: seq(node):: ispath(*$x \circ p \circ y$*)* $\wedge \forall$ *z: node::* $z \in p \Rightarrow z \leq n$ (unfold)
- = $\exists$ *P: seq(node):: ispath(*$x \circ p \circ y$*)* (simplification: $z \leq n$ holds for all nodes $z$)
- = *trans(x,y)* (fold).

To obtain an executable function from the given specification, a next eureka-step is the introduction of recursion. This is done by formulating a base case (for $i = 0$) and a recursive case:

- *Lemma 2:* $tc(0, x, y) = edge(x, y)$
- *Lemma 3:* $tc(i + 1, x, y) = tc(i, x, y) \vee [tc(i, x, i + 1) \wedge tc(i, i + 1, y)]$.

The proof of lemma 2 is analogous to that of lemma 1. The proof of lemma 3 involves an additional case-analysis (all nodes on a path are smaller or equal to node $i$ vs. a path contains a node $i + 1$) and application of logical rules.

The proved lemmas give rise to the program

- **function** *tc(i:nat, x:node, y:node):bool;*
  *if* $i = 0$
  *then edge(x,y)*
  *else tc(i,x,y)* $\vee$ *[tc(i,x,i+1)* $\wedge$ *tc(i,i+1,y)].*

This example demonstrates, how the development of a program from a non-executable initial specification can be supported by CIP. Program development was mainly "intuition-guided", but the transformation system supports the programmer in two ways: First, the notion of program transformation results in systematic program development by stepwise refinement, and second, the transformation system (partially) automatically generates proofs for the correctness of the programmers intuitions.

Given the recursive function *tc*, in a next step it can be transformed from its inefficient recursive realization into a more efficient version realized by *for*-loops. This transformational implementation can be performed automatically by means of sophisticated (and verified) transformation rules.

### 2.2.3 PROGRAM SYNTHESIS BY STEPWISE REFINEMENT: KIDS

KIDS[4] is currently the most successful system for deductive (transformational) program synthesis. KIDS supports the development of correct and efficient programs by applying consistency-preserving transformations to an initial specification, first transforming the specification into an executable but inefficient program and then optimizing this program. The basis of KIDS is *Refine* – a wide-spectrum language for representing specifications as well as programs. The final program is represented in Common Lisp.

In KIDS a variety of state-of-the-art techniques are integrated into one system and KIDS makes use of a large, formalized amount of domain and programming knowledge: It offers program schemes for divide-and-conquer, global search (binary search, depth-first search, breadth-first search), and local search (hill climbing), it makes use of a library of reusable domain axioms (equations specifying abstract data types), it integrates deductive inference (with about 500 inference rules represented as directed transformation rules), and it includes a variety of state-of-the-art techniques for program optimization (such as expression simplification and finite differencing).

KIDS is an interactive system where the user typically goes through the following steps of program development (Smith, 1990):

**Develop a Domain Theory:** The user defines types and functions and provides laws that allow high-level reasoning about the defined functions. The user can develop the theory from the scratch or make use of a hierarchic library of types.

For example, a domain theory for the $k$-Queens problem gives boolean functions representing the constraints that no two queens can be in the same row or column and that no two queens can be in the same diagonal of a chessboard. Laws for reasoning about functions typically include monotonicity and distributive laws.

**Create a Specification:** The user enters a specification statement in terms of the domain theory.

For example, a specification of queens states that for $k$ queens on a $k \times k$ board, a set of board positions must be returned for which the constraints defined in the domain theory hold.

**Apply a Design Tactic:** The user selects a program scheme, representing a tactic for algorithm design, and applies it to the specification. This is the crucial step of program development – transforming a typically non-executable specification into an (inefficient) high-level program.

[4] see `http://www.kestrel.edu/HTML/prototypes/kids.html`

For example, the $k$-Queens specification can be solved with global search, which can be refined to depth-first search (find legal solutions using backtracking).

**Apply Optimizations:** The user selects an optimization operation and a program expression to which it should be applied. The optimization techniques are fully automatic.

**Apply Data Type Refinements:** The user can select implementations for the high-level data types in the program.
Which implementation of a data type will result in the most efficient program depends on the kind of operations to be performed. For example, sets could be realized as lists, arrays, or trees.

**Compile:** The code is compiled into machine-executable form (first Common Lisp and then machine code).

The $k$-Queens example is given in detail in Smith (1990). The idea of program development by stepwise refinement of abstract schemes is also illustrated in Smith (1985) for constructing a divide-and-conquer based search algorithm.[5]

The KIDS system demonstrates that it is possible that automatic program synthesis can cover all steps of program development from high-level specifications to efficient programs. Furthermore, it gives some evidence to the claim of KBSE that automatization can result in a higher level of productivity in software development. High productivity is also due to the possibility of reuse of domain axioms. But of course, the system is not "auto-magic" – it strongly depends on interactions with an *expert* user, especially for the initial development steps from domain theory to selection of a suitable design tactic. These first steps can be seen as an approach to meta-programming, similar to CIP. Even for optimization, the user must have knowledge which parts of the program could be optimized in what way to select the appropriate expressions and rules to be applied to them.

### 2.2.4 CONCLUDING REMARKS

In principle, there is no fundamental difference between constructive theorem proving and program transformation: In both cases, starting point is a formal specification and output is a program which is correct with respect to the specification. Theorem proving can be converted into transformation if axioms are represented as rules (equations may correspond to bi-directional rules) and if proof-rules (such as resolution) are represented as special rewrite rules. The

[5]Here the system *Cypress*, which was the predecessor of KIDS, was used.

advantages of transformation over proof systems for program synthesis are, that forward reasoning and a smaller formal overhead makes program derivation somewhat easier (Kreitz, 1998). In contrast, proof systems are typically used for (ex-post) program *verification*.

As described above, program transformation systems might include non-verified rules representing domain knowledge which are applied in interaction with the system user. Such rules are mainly applied during the first steps of program development until an initial executable program is constructed. The following optimization steps (transformational programming) mainly rely on verified rules which can be applied fully automatically. Code optimization by transformation is typically dealt with in text books on compilers (Aho, Sethi, and Ullman, 1986). A good introduction to program transformation and optimization is given in Field and Harrison (1988).

An interesting approach to "reverse" transformation – from efficient loops (tail recursion) to more easily verifiable (linear) recursion – is proposed by Giesl (2000).

# 3 INDUCTIVE APPROACHES

In the following we introduce inductive approaches to program synthesis. First, basic concepts of inductive inference are introduced. Inductive inference is researched in philosophy (epistemology), in cognitive psychology, and in artificial intelligence. In this section we focus on induction in AI, that is, on machine learning. In chapter 9 we discuss some relations between inductive program synthesis and inductive learning of strategies in human problem solving. Grammar inference is introduced as the theoretical background for program synthesis. In the following sections, three different approaches to program synthesis are described – genetic programming, inductive logic programming, and synthesis of functional programs. Synthesis of functional programs is the oldest and genetic programming the newest approach. In genetic programming hypothesis construction relies strongest on search in hypotheses-space, while in functional program synthesis hypothesis construction is most strongly guided by the structure of the examples.

## 3.1 FOUNDATIONS OF INDUCTION

As introduced in section 1.2.2, induction means the inference of generalized rules from examples. The ability to perform induction is a presupposition for learning – often induction and learning are used as synonyms.

### 3.1.1 BASIC CONCEPTS OF INDUCTIVE LEARNING

Inductive learning can be characterized in the following way (Mitchell, 1997, p. 2):

**Definition 6.10 (Learning)** *A computer program is said to learn from experience $E$ with respect to some class of tasks $T$ and performance measure $P$, if its performance at tasks in $T$, as measured by $P$, improves with experience $E$.*

In the following, we will flesh-out this definition.

**Learning Tasks.** Learning tasks can be roughly divided into classification tasks versus performance tasks, labeled as *concept learning* versus *skill acquisition*. Examples for concept learning are recognizing handwritten letters, or identifying dangerous substances. Examples for skill acquisition are navigating without bumping into obstacles, or controlling a chemical process such that certain variables stay in a pre-defined range.

In inductive program synthesis, the learning task is to construct a program which transforms input values from a given domain into the desired output values. A program can be seen as representation of a concept: Each input value is classified according to the operations which transform it into the output value. Especially, programs returning boolean values, can be viewed as concept definitions. Typical examples are recursive definitions of *odd* or *ancestor* (see fig. 6.1). A program can also be seen as representation of a (cognitive) skill: The program represents such operations which must be performed for a given input value to fulfill a certain goal. For example, a program *quicksort* represents the skill of efficiently sorting lists; or a program *hanoi* represents the skill of solving the Tower of Hanoi puzzle with a minimum number of moves (see fig. 6.1).

**Learning Experience.** Learning experience typically is provided by presenting a set of *training examples*. Examples might be pre-classified, then we speak of *supervised learning* or learning with a teacher, otherwise we speak of *unsupervised learning* or learning from observation.

For example, for learning a single concept – such as dangerous versus harmless substance – the system might be provided with a number of chemical descriptions labeled as positive (i. e., dangerous) and negative (i. e., harmless) instances of the to be learned concept. For learning to control a chemical process, the learning system might be provided with current values of a set of control variables, labeled with the appropriate operations which must be performed in that case and each operation sequence constitutes a class to be learned. Alternatively, for skill acquisition a learning system might provide its own experience, performing a sequence of operations and getting feedback for the final outcome (reinforcement learning). In this case, the examples are labeled only indirectly – the system is confronted with the credit assignment problem, determining the degree to which each performed operation is responsible for the final outcome.

```
; functional program deciding whether a natural number is odd
odd(x) = if (x = 0) then false else if (x = 1) then true else odd(x-2)

; logical program deciding whether p is ancestor of f
ancestor(P,F) :- parent(P,F).
ancestor(P,F) :- parent(P,I), ancestor(I,F).

; logical program for quicksort
qsort([X|Xs],Ys) :-
  partition(Xs,X,S1,S2),
  qsort(S1,S1s),
  qsort(S2,S2s),
  append(S1s,[X|S2s],Ys).
qsort([],[]).

; functional program for hanoi
hanoi(n,a,b,c) = if (n = 1) then move(a,b)
                            else append(hanoi((n-1),a,c,b), move(a,b),
                                        hanoi((n-1),c,b,a))
```

*Figure 6.1.* Programs Represent Concepts and Skills

In the simplest form, examples might consist just of a set of attributes. Such attributes might be categorial (substance contains fluor yes/no) or metric (current temperature has value $x$). In general, examples might be structured objects, that is sets of relations or terms. For instance, for the *ancestor* problem (see fig. 6.1) kinship between different persons constitute the learning experience. In program synthesis, examples are always structured objects.

Training examples can be presented stepwise (*incremental learning*) or all at once ("batch" learning). Both learning modes are possible for program synthesis. Finally, training examples should be a representative sample of the concept or skill to be learned. Otherwise, the performance of the learning system might be only successful for a subset of possible situations.

In table 6.2 possible training examples for the *ancestor* and the *hanoi* problems are given. In the first case, given some parent-child relations, positive and negative examples for the concept "ancestor" are presented. In the second case, examples are associated with sequences of *move* operations for different numbers of discs. The *hanoi* example can be classified as input to supervised or unsupervised learning: The number of discs ($n = i, i = 1 \ldots 3$) represents a possible input and the *move* operations represent the class, the input must be associated with (see Briesemeister et al., 1996). Alternatively, the disc-number/operations pairs can be seen as patterns and the learning task is to extract their common structure (descriptive generalization). While in the first case, the different operator sequences constitute different classes to be learnt,

*Table 6.2.* Training Examples

```
; ANCESTOR(X,Y)
parent(peter,robert)    parent(mary,robert)
parent(robert,tim) parent(tim,anna)
parent(tim,john) parent(julia,john)
... ...

; pre-classified positive and negative examples
ancestor(julia,john)   not(ancestor(anna,tim))
ancestor(peter,tim)   not(ancestor(anna,mary))
ancestor(peter,anna)   not(ancestor(john,robert))
...   ...

; HANOI(N,A,B,C)
(N = 1) --> (move(A,B))
(N = 2) --> (move(A,C),move(A,B),move(C,B))
(N = 3) --> (move(A,B),move(A,C),move(B,C),
             move(A,B),move(C,A),move(C,B),
             move(A,B))
```

in the second case, all examples are positive. The examples are ordered with respect to the number of discs. Therefore, in an indirect way, they also contain information about what cases are not belonging to the domain.

**Additional Information.** The learning system might be provided with additional information besides the training examples. For instance, in the *ancestor* problem in table 6.2 some parent-child relations are directly given. These facts are essential for learning: It is not possible to come up with a terminating recursive rule (as given in fig. 6.1) without reference to a direct relation as *parent*.

If a system is given additional information besides the training examples, this is typically called *background knowledge*. Background knowledge can have different functions for learning: it can be necessary for hypothesis construction, it can make hypothesis construction more efficient, or it can make the resulting hypothesis more compact or readable.

The *ancestor* example could have been presented in a different way, as shown in table 6.3. Instead of giving *parent* relations, *mother* and *father* relations can be used, together with a rule stating, that *father(X,Y)* or *mother(X,Y)* implies *parent(X,Y)*. Another example for using background knowledge for learning, is to give pre-defined rules for *append* and *partition* for learning *quicksort* (see fig. 6.1).

*Table 6.3.* Background Knowledge

```
; ANCESTOR(X,Y)
father(peter,robert)    mother(mary,robert)
father(robert,tim)      father(tim,anna)
father(tim,john)        mother(julia,john)
...
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
```

Additional information can also be provided in form of an *oracle*: The learning system constructs new examples which are consistent with its current hypothesis and asks an oracle whether the example belongs to the class to be learned. An oracle is usually the teacher.

**Representing Hypotheses.** The task of the learning system is to construct generalized rules (representing concepts or skills) from the training examples and background knowledge. In other words, learning means to construct an intensional representation from a sample of an extensional representation of a concept or skill. As mentioned in section 1.2.2, such generalized rules have the status of hypotheses.

For supervised concept learning, a hypothesis is a special case of a function $f : X \rightarrow Y$ which maps a given object belonging to the learning domain to a class. In general, $f$ can be a linear or a non-linear function, mapping a vector of attribute values $X$ into an output value $Y$, representing the class the object described by this vector is supposed to belong to. Alternatively, hypotheses can be represented symbolically – the most prominent example are decision trees.

In the context of program synthesis, inputs $X$ are structural descriptions (relations or terms), outputs $Y$ are operator sequences transforming the input into the desired output, and hypotheses are represented as logical or functional programs (see fig. 6.1): For the *ancestor* problem, function $f(x)$ is represented by clauses *ancestor(X,Y)* which return *true* or *false* for a given set of parent-child relations and two persons. For the *hanoi* problem, function $f(x)$ is represented by a functional program which returns a sequence of *move*-operations to transport a tower of $n$ discs from the start peg to the goal peg.

**Identification Criteria and Performance Measure.** There are two stages in which the learning system is evaluated: First, a decision criterium is needed to terminate the learning algorithm, and, second, the quality of the learned hypothesis must be tested. An algorithm terminates, if some identification criterium with respect to the training examples is fulfilled. The classical concept here is *identification in the limit*, proposed by Gold (1967). The learner

terminates if the current hypothesis holds at some point during learning for all examples. An alternative concept is PAC (probably approximately correct) identification (Valiant, 1984). The learner terminates if the current hypothesis can be guaranteed with a high probability to be consistent with future examples.

The first model represents an all-or-nothing perspective on learning – a hypothesis must be totally correct with respect to the seen examples. The second model is based on probability theory. Most approaches of concept learning are based on PAC-learnability, while program synthesis typically is based on Gold's theory (see sect. 3.1.2). An analysis of program synthesis from traces in the PAC theory was presented by Cohen (1995, 1998).

To evaluate the quality of the learned hypothesis, the learning system is presented with new examples and generalization accuracy is obtained. Typically, the same criterium is used as for termination.

**Learning Algorithm.** A learning algorithm gets training examples and background knowledge as input and returns a hypothesis as output.

The majority of learning algorithms is for constructing non-recursive hypotheses for supervised concept learning where examples are represented as attribute vectors (Mitchell, 1997). Among the most prominent approaches are function approximation algorithms– such as perceptron, back-propagation, or support vector machines –, Bayesian learners, and decision tree algorithms. For unsupervised concept learning, the dominant approach is cluster analysis.

Approaches for learning recursive hypotheses from structured examples – that is, approaches to inductive program synthesis – are genetic programming, inductive logic programming, and synthesis of functional programs. These approaches will be described in detail below.

In general, a learning algorithm is a search-algorithm in the space of possible hypotheses. There are two main learning mechanisms: *data-driven* approaches start with the most specific hypothesis, that is, the training examples, and *approximation-driven* (or model-driven) approaches start with a set of approximate (incomplete, incorrect) hypotheses. Data-driven learning is incremental, iterating over the training examples (Flener, 1995). Each iteration returns a hypothesis which might be the searched for final hypothesis. Approximation-driven learning is non-incremental. Current hypotheses are generalized if not all positive examples are covered, or specialized if other than the positive examples are covered.

**Induction Biases.** As usual, there is a trade-off between the expressiveness of the hypotheses language and the efficiency of the learning algorithm (cf.,. the discussion of domain specification languages and efficiency of planning algorithms, chap. 2). Typically, in machine learning search is restricted by so called induction biases (Flener, 1995, pp. 33): Each learning algorithm is

restricted by a syntactic bias, that is, a restriction of the hypothesis language. For logical program synthesis, the language can be restricted to a subset of Prolog clauses. For functional program synthesis, the form of functions can be restricted by program schemes. Furthermore, a semantic bias might restrict the vocabulary available for the hypothesis. For example, when synthesizing Lisp programs, the language primitives might be restricted to *car*, *cdr*, *cons*, and *atom* (see sect. 3.4).

**Machine Learning Literature.** The AI text books of Winston (1992) and Russell and Norvig (1995) include chapters on machine learning techniques. An excellent text book on machine learning was written by (Mitchell, 1997). Collections of important research papers, including work on inductive program synthesis, are presented by Michalski, Carbonell, and Mitchell (1983) and Michalski, Carbonell, and Mitchell (1986). A collection of papers on knowledge acquisition and learning was edited by Buchanan and Wilkins (1993). Flener (1995) discusses machine learning in relation to program synthesis.

### 3.1.2 GRAMMAR INFERENCE

Most of the basic concepts of machine learning as introduced above have their origin in the work of Gold (1967). Gold modeled learning as the acquisition of a formal grammar from example and provided a classification of models of language learnability together with fundamental theoretical results. Grammar inference research developed from Gold's work in two directions – providing theoretical results of learnability for certain classes of languages given a certain kind of learning model and recently also the development of efficient learning algorithms for some classes of formal grammars. We focus on the conceptual and theoretical aspect of grammar inference, that is on algorithmic learning theory. Learning theory can be seen as a special case of complexity theory where the complexity of languages is researched with respect to the effort of their enumeration (by application of rules of a formal grammar) or with respect to the effort of their identification by an automaton.

**Definition 6.11 (Formal Language)** *For a given non-empty finite set $A$, called alphabet of the language, $A^*$ represents the set of all finite strings over elements from $A$. A language $L$ is defined as: $L \subseteq A^*$.*

Learning means to assign a grammar (called *naming relation* by Gold) to a language $L$ after seeing a sequence of example strings:

**Definition 6.12 (Language Learning)** *For a sequence of time steps, the learner is at each time step confronted with a unit of information $i_t$ concerning the unknown language $L$. A stepwise presentation of units $i$ is called* training

sequence. *At each time step, the learner makes a guess $g_t$ of the grammar characterizing $L$, based on the information it has received from the start of learning to time step $t$. That is, the learner is a function $G$ with $g_t = G(i_1, \ldots, i_t)$.*

As introduced above, Gold introduced the concept of "identification in the limit" to define the learnability of a language:

**Definition 6.13 (Identification in the Limit)** *$L$ is* identified in the limit *if after a finite number of time steps the guesses are always the same, that is, $g_t = g_{t+i}, i \geq 0$. A class of languages $L$ is called* identifiable in the limit *if there exists an algorithm such that each language of the class will be identified in the limit for any allowable training sequence.*

Identification in the limit is dependent on the *method of information presentation*. Gold discerns presentation of text and learning with an informant. Learning from text means learning from positive examples only, where examples are presented in an arbitrary order. Learning with informant corresponds to supervised learning. The informant can present positive and negative examples and can provide some methodical enumeration of examples. Gold investigates three modes of learning from text and three modes of learning with informant:

- Method of information presentation: **Text**
  A text is a sequence of strings $x_1, x_2, \ldots$ from $L$ such that each string of $L$ occurs at least once. At time $t$ the learner is presented $x_t$.
  - Arbitrary Text: $x_t$ may be any function of $t$.
  - Recursive Text: $x_t$ may be any recursive function of $t$.
  - Primitive Recursive Text: $x_t$ may be any primitive recursive function of $t$.
- Method of information presentation: **Informant**
  An informant for $L$ tells the learner at each time $t$ whether a string $y_t$ is element of $L$. There are different ways of how to choose $y_t$
  - Arbitrary Informant: $y_t$ may be any function of $t$ as long as every string of $A^*$ occurs at least once.
  - Methodical Informant: An enumeration is assigned a priori to the strings of $A^*$ and $y_t$ is the $t$-th string of the enumeration.
  - Request Informant: At time $t$ the learner chooses $y_t$ on the basis of information received so far.

The method "request informant" is currently an active area of research called active learning (Thompson, Califf, and Mooney, 1999). Gold showed that all three methods of information presentation by informant are equivalent

(Theorem I.3, in Gold, 1967). Obviously, it is a harder problem to learn from text only than to learn from an informant, because in the second case not only positive but also negative examples are given.

Identification in the limit is also dependent on the "naming relation", that is, on the way, in which a hypothesis about language $L$, that is a grammar, is represented. The two possible naming relations for languages are

- **Language Generator**: A Turing machine which generates $L$.
  A generator is a function from positive integers (corresponding to time steps $t$) to strings in $A^*$ such that the function range is exactly $L$. A generator exists iff $L$ is recursively enumerable.
- **Language Tester**: A Turing machine which is a decision procedure for $L$.
  A tester is a function from strings to $\{0, 1\}$ with value 1 for strings in $L$ and value 0 otherwise.

Generators can be transformed in testers but not the other way round and it is simpler to learn a generator for a class of languages than a tester (Gold, 1967).

The definition of learnability as identification in the limit together with a method of information presentation and a naming relation constitutes a *language learnability model.*

Gold presented some methods for identification in the limit, the most prominent one is *identification by enumeration.*

**Definition 6.14 (Identification by enumeration[6])** *Let $D$ be a description for a class of languages where the elements of $D$ are effectively enumerable. Let $L(d)$ be the language denoted by description $d$. For each time step $t$, search for the least $k$ such that $i_t$ is in $L(d_k)$ and that all preceding $i_j, j = 1 \ldots t-1$ are in $L(d_k)$, too.*

Identification by enumeration presupposes a linear ordering of possible hypotheses. Each incompatibility between the training examples seen so far and the current hypothesis results in the elimination of this hypothesis. That is, if hypotheses are effectively enumerable, it is guaranteed that the $t$-th guess is the earliest description compatible with the first $t$ elements of the training sequence and that learning will converge to the first description of $L(d)$ in the given enumeration $D$. In practice, enumeration is mostly not efficient. It is often possible, to organize the class of hypotheses in a better way such that whole subclasses can be eliminated after some incompatibility is identified (Angluin, 1984). The notion of inductive inference as search though a given space of hypotheses is helpful for theoretical considerations. In practice, this space typically is only given indirectly by the inference procedure and hypotheses are *generated* and tested at each inference step.

*Table 6.4.* Fundamental Results of Language Learnability (Gold, 1967, tab. 1)

| Learnability Model | Class of Languages |
|---|---|
| Primitive Recursive Text + Generator Naming | |
| | Recursively enumerable recursive |
| Informant | |
| | Primitive recursive |
| | Context-sensitive |
| | Context-free |
| | Regular |
| | Superfinite |
| Text | |
| | Finite cardinality languages |

Gold (1967) provided fundamental results about which classes of languages are identifiable in the limit given which learnability model. These results are summarized in table 6.4. Later work in the area of learning theory is mostly concerned with refining Gold's results by identifying interesting subclasses of the original categorization of Gold and providing analyses of their learnability together with efficiency results (Zeugmann and Lange, 1995; Sakakibara, 1997).

Theorems and proofs concerning all results given in table 6.4 can be found in the appendix of Gold (1967). We just highlight some results, presenting the theorems for the class of finite cardinality languages (languages with a finite number of words) and for the class of superfinite languages, containing all languages of finite cardinality and at least one of infinite cardinality.

**Theorem 6.1 (Learnability of Finite Cardinality Languages)**
*(Gold, 1967, theorem I.6) Finite cardinality languages are identifiable from (arbitrary) text, that is, from positive examples presented in some arbitrary sequence, only.*

**Proof 6.1 (Learnability of Finite Cardinality Languages)**
*Imagine a language which consists of integers from* $1$ *to a fixed number* $n$*. For any information sequence* $i_1, i_2, \ldots$ *the learner can assume that the language consists only of the numbers seen so far. Since* $L$ *is finite, after some time all elements of* $L$ *have occured in the information sequence, such that the guess will be correct.*

It is enough to give the proof for a language consisting of a finite set of positive integers because all other finite languages can be mapped on this language.

**Theorem 6.2 (Learnability of Superfinite Languages)**
*(Gold, 1967, theorem I.9) Using information presentation by recursive text and the generator-naming relation, any class of languages which contains all finite languages and at least one infinite language $L$ is not identifiable in the limit.*

**Proof 6.2 (Learnability of Superfinite Languages)**
*The infinite language $L$ is recursively enumerable (because otherwise it would not have a generator). It can be shown that there exists a recursive sequence of positive integers which ranges over $L$ without repetitions. If the information sequence is presented as recursive sequence $a_1, a_2, \ldots$ then at some time step, this information sequence is a recursive text for the finite language $L = \{a_1, \ldots a_t\}$. At a later time step $t'$ some additional element of the infinite language can be presented and the information sequence now is a recursive text for the finite language $L' = \{a_1, \ldots, a_t, a_{t'}\}$. From this observation follows, that the learning system must change its guess an infinite number of times.*

Primitive recursive languages are learnable from an informant because for such languages exists an efficient enumeration (Hopcroft and Ullman, 1980, chap. 7). Because learning with informant includes the presentation of negative examples, these can be used to remove incompatible hypothesis from search. While primitive recursive language are identifiable in the limit, it is not possible – even for regular languages – to find a minimal description (a deterministic finite automaton with a minimal number of states) in polynomial time (Gold, 1978)!

Most of research in grammar inference is concerned with regular languages and subclasses of them (as pattern languages). Angluin (1981) could show that a minimal deterministic finite automaton can be inferred in polynomial time from positive and negative examples, if additionally to an informant, an oracle answering membership queries is used. Angluin's ID algorithm is used for example by Schrödl and Edelkamp (1999) for inferring recursive functions from user-generated traces. Boström (1998) showed how discriminating predicates can be inferred from only positive examples using an algorithm for inferring a regular grammar.[7]

In chapter 7 we will show that the class of recursive programs addressed by our approach to folding finite ("initial") programs corresponds to context-free tree grammars. Finite programs can be seen as primitive recursive text and folding, that is, generating a recursive program which results in the given finite program as its $i$-th unfolding correspond to generator-naming (see first line in tab. 6.4).

[7] More information about grammar inference, including algorithms and areas of application, can be found via `http://www.cs.iastate.edu/~honavar/gi/gi.html`.

## 3.2 GENETIC PROGRAMMING

Genetic programming as an evolutionary approach to program synthesis was established by Koza (1992). Starting point is a population of randomly generated computer programs. New programs – better adapted to the "environment", which is represented by examples together with an "fitness" function – are generated by iterative application of Darwinian natural selection and biologically inspired "reproduction" operations.[8] Genetic programming is applied in system identification, classification, control, robotics, optimization, game playing, and pattern recognition.

### 3.2.1 BASIC CONCEPTS

**Representation and Construction of Programs.** In genetic programming, typically *functional* programs are synthesized. Koza (1992) synthesizes Lisp functions, an application of genetic programming to the synthesis of ML functions was presented by Olsson (1995). These programs are represented as trees. A program is constructed as a syntactically correct expression over a set of predefined functions and a set of terminals. The set of functions can contain standard operators (e. g., arithmetic operators, boolean operators, conditional operators) as well as user-defined domain specific functions. For each function, its arity is given. The set of terminals contains variables and atoms. Atoms represent constants. Variables represent program arguments. They can be instantiated by values obtained for example from "sensors".

A random generation of a program typically starts with an element of the set of functions. The function becomes the root node in the program tree and it branches in the number of arcs given by its arity. For each open arc an element from the set of functions or terminals is introduced. Program generation terminates if each current node contains a terminal. Examples for program trees are given in figure 6.2.

**Representation of Problems.** A problem together with a "fitness" measure constitutes the "environment" for a program. Problems are represented by a set of examples. For some problems, this can be pairs of possible input values, together with their associated outputs. The searched for program must transform possible inputs into desired outputs. Fitness can be calculated as the number of erroneous outputs generated by a program. Such problems can be seen as typical for inductive program synthesis. For learning to play a game successfully, the problem can consist of a set of possible constellations together with a scoring function. The searched for program represents a strategy for playing this game. For each constellation, the optimal move must be selected.

[8]The genetic programming home page is `http://www.genetic-programming.org/`.

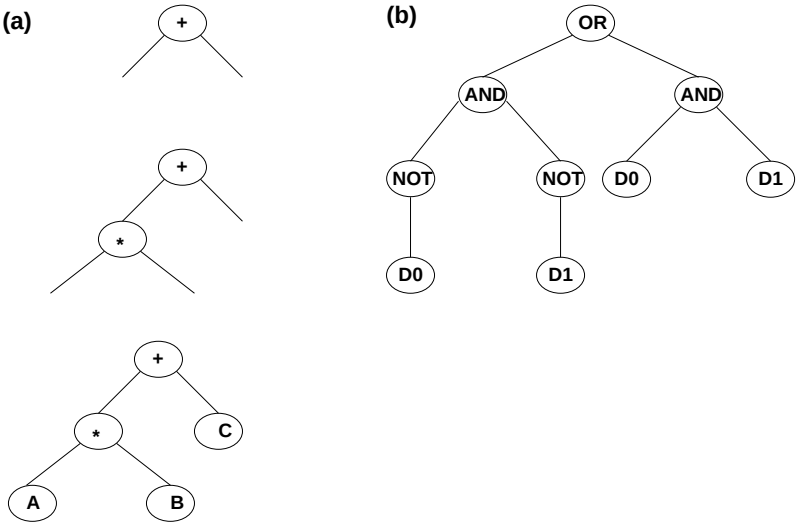



*Figure 6.2.* Construction of a Simple Arithmetic Function (a) and an Even-2-Parity Function (b) Represented as a Labeled Tree with Ordered Branches (Koza, 1992, figs. 6.1, 6.2)

Fitness can for example be calculated as the percentage of wins. For learning a plan, the input consists of a set of possible problem states. The searched for program generates action sequences to transform a problem state into the given goal state. Fitness can be calculated as the percentage of problem states for which the goal state is reached. An overview of a variety of problems is given in Koza (1992, tab. 2.1).

**Generational Loop.** Initially, thousands of computer programs are randomly generated. This "initial population" corresponds to a blind, random search in the space of the given problem. The size of the search space is dependent on the size of the set of functions over which programs can be constructed. Each program corresponds to an "individual" whose "fitness" influences the probability with which it will be selected to participate in the various "genetic operations" (described below). Because selection is not absolutely dependent on fitness, but fitness just influences the probability of selection, genetic programming is not a purely greedy-algorithm. Each iteration of the generational loop results in a new "generation" of programs. After many iterations, a program might emerge which solves or approximately solves the given problem. An abstract genetic programming algorithm is given in table 6.5.

**Genetic Operations.** The set of genetic operations are:

*Table 6.5.* Genetic Programming Algorithm (Koza, 1992, p. 77)

1. Generate an initial population of random compositions of the functions and terminals of the problem.
2. Iteratively perform the following sub-steps until the termination criterium has been satisfied:
   (a) Execute each program in the population and assign it a fitness value according to how well it solves the problem.
   (b) Select computer program(s) from the current population chosen with a probability based on fitness.
   Create a new population of computer programs by applying the following two primary operations:
      i. Copy program to the new population (Reproduction).
      ii. Create new programs by genetically recombining randomly chosen parts of two existing programs.
3. The best so-far individual (program) is designated as result.

**Mutation:** Delete a subtree of a program and grow a new subtree at its place randomly.
This "asexual" operation is typically performed sparingly, for example with a probability of 1% during each generation.

**Crossover:** For two programs ("parents"), in each tree a cross-over point is chosen randomly and the subtree rooted at the cross-over point of the first program is deleted and replaced by the subtree rooted at the cross-over point of the second program.
This "sexual recombination" operation is the predominant operation in genetic programming and is performed with a high probability (85% to 90 %).

**Reproduction:** Copy a single individual into the next generation.
An individuum "survives" with for example 10% probability.

**Architecture Alteration:** Change the structure of a program.
There are different structure changing operations which are applied sparingly (1% probability or below):

**Introduction of Subroutines:** Create a subroutine from a part of the main program and create a reference between the main program and the new subroutine.

**Deletion of Subroutines:** Delete a subroutine; thereby making the hierarchy of subroutines narrower or shallower.

**Subroutine Duplication:** Duplicate a subroutine, give it a new name and randomly divide the preexisting calls of the subroutine between the old

and the new one. (This operation preserves semantics. Later on, each of these subroutines might be changed, for example by mutation.)

**Argument Duplication:** Duplicate an argument of a subroutine and randomly divide internal references to it. (This operation is also semantics preserving. It enlarges the dimensionality of the subroutine.)

**Argument Deletion:** Delete an argument; thereby reducing the amount of information available to a subroutine ("generalization").

**Automatically Defined Iterations/Recursions:** Introduce or delete iterations (ADIs) or recursive calls (ADRs).
Introduction of iterations or recursive calls might result in non-termination. Typically, the number of iterations (or recursions) is restricted for a problem. That is, for each problem, each program has a time-out criterium and is terminated "from outside" after a certain number of iterations.

**Automatically Defined Stores:** Introduce or delete memory (ADSs).

Quality criteria for programs synthesized by genetic programming are correctness – which is defined as 100% fitness for the given examples –, efficiency and parsimony. The two later criteria can be additionally coded in the fitness measure. In the following, we give examples of program synthesis by genetic programming.

### 3.2.2 ITERATION AND RECURSION

First we describe, how a plan for stacking blocks in a predefined order can be synthesized where the final program involves iteration of actions (Koza, 1992, chap. 18.1). This problem is similar to the *tower* problem described in section 1.4.5: An initial problem state consists of $n$ blocks which can be stacked or lying on the table. Problem solving operators are *puttable(x)* and *put(x,y)*, the first operator defining how a block $x$ is moved from another block on the table and the second operator defining how a block $x$ can be put on top of another block $y$. The problem solving goal defines in which sequence the $n$ blocks should be stacked into a single tower (see fig. 6.3).

The set of terminal is $T = \{CS, TB, NN\}$ and the set of functions is $\{MS, MT, NOT, EQ, DU\}$ with

***CS*:** A sensor that dynamically specifies the top block of the *Stack*.

***TB*:** A sensor that dynamically specifies the block in the *Stack* which together with all blocks under it are already positioned correctly. ("top correct block")

***NN*:** A sensor that dynamically specifies the block which must be stacked immediately on top of *TB* according to the goal. ("next needed block")

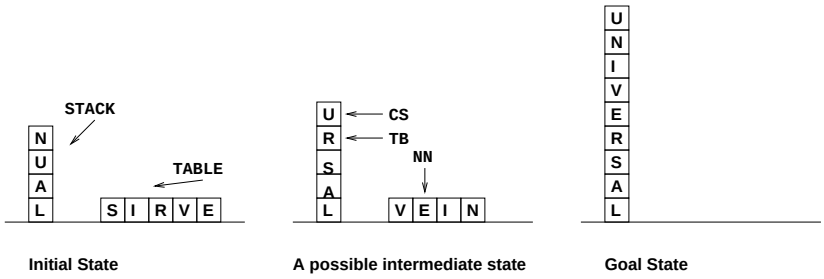



*Figure 6.3.* A Possible Initial State, an Intermediate State, and the Goal State for Block Stacking (Koza, 1992, figs. 18.1, 18.2)

***MS*:** A move-to-stack operator with arity one which moves a block from the *Table* on top of the *Stack*.

***MT*:** A move-to-table operator with arity one which moves a block from the top of the *Stack* to the *Table*.

***NOT*:** A boolean operator with arity one switching the truth value of its argument.

***EQ*:** A boolean operator with arity two which returns true if its arguments are identical and false otherwise.

***DU*:** A user-defined iterative "do-until" operator with arity two. The expression `DU *Work* *Predicate*` causes `*Work*` to be iteratively executed until `*Predicate*` is satisfied.

All functions have defined outputs for all conditions: *MS* and *MT* change the *Table* and *Stack* as side-effect. They return *true*, if the operator can be applied successfully and *nil* (*false*) otherwise. The return value of *DU* is also a boolean value indicating whether `*Predicate*` is satisfied or whether the *DU* operator timed out.

For this example, the set of terminals and the set of functions are carefully crafted. Especially the pre-defined sensors carry exactly that information which is relevant for solving the problem! The problem is more restricted than the classical *blocks-world* problem: While in the blocks-world, any constellation of blocks (for example, four stacks containing two blocks each and one with only one block) is possible (see tab. 2.3 for the growth of the number of states in dependence of the number of blocks), in the block-stacking problem, only one stack is allowed and all other blocks are lying on the table.

For the evolution of the desired program 166 fitness cases were constructed: ten cases where zero to all nine blocks in the stack were already ordered correctly; eight cases where there is one out of order block on top of the

Population Size: M = 500
Fitness Cases: 166

**Correct Program:**
Fitness: *166 - number of correctly handled cases* Termination: Generation 10

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

**Correct and Efficient Program:**
Fitness: $0.75 \cdot C + 0.25 \cdot E$ with
$C =$ ( *number of correctly handled cases*$/166) \cdot 100$ $E = f(n)$ *as function of the total number of moves over all 166 cases:*
*with* $f(n) = 100$ *for the analytically obtained minimal number of moves for a correct program (*$min = 1641$*);*
$f(n)$ *linearly scaled upwards for zero moves up to 1640 moves with* $f(0) = 0$
$f(n)$ *linearly scaled downwards for 1642 moves up to 2319 moves (obtained by the first correct program) and* $f(n) = 0$ *for* $n > 2319$
Termination: Generation 11

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

**Correct, Efficient, and Parsimonious Program:**
Fitness: $0.70 \cdot C + 0.20 \cdot E + 0.10 \cdot$ *(number of nodes in program tree)*
Termination: Generation 1

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

*Figure 6.4.* Resulting Programs for the Block Stacking Problem (Koza, 1992, chap. 18.1)

stack; and a random sampling of 148 additions cases. Fitness was measured as the number of correctly handled cases. Koza (1992) reports three variants for the synthesis of a block stacking program, summarized in figure 6.4. The first, correct program first moves all blocks on the table and than constructs the correct stack. This program is not very efficient because there are made unnecessary moves from the stack to the table for partially correct stacks. Over all 166 cases, this function generates 2319 moves in contrast to 1642 necessary moves. In the next trial, efficiency was integrated into the fitness measure and as a consequence, a function calculating only the minimum number of moves emerged. But this function has an outer loop which is not necessary. By integrating parsimony into the fitness measure, the correct, efficient, and parsimonious function is generated.

As an example for synthesizing a recursive program Koza (1992, chap. 18.3) shows how a function calculation Fibonacci numbers can be generated from examples for the first twenty numbers. The problem is treated as sequence

*Table 6.6.* Calculation the *Fibonacci* Sequence

```
(+ (SRF (- J 2) 0)
   (SRF (+ (+ (- J 2) 0) (SRF (- J J) 0))
        (SRF (SRF 3 1) 1)))
```

induction problem – for an ascending order of index positions $J$, the function $f(J)$ must be detected. As terminal set $T = \{J, 0, 1, 2, 3\}$ is given and as function set $F = \{+, -, *, SRF\}$. Terminal $J$ represents the input value. Function *SRF* is a "sequence referencing function" with `(SRF K D)` returns either the Fibonacci number for $K$ or a default value $D$. The used strategy is, that the Fibonacci number are calculated for $K = 0 \ldots 20$ in ascending order and each result is stored in a table. Such, *SRF* has access to the Fibonacci numbers for all $K$ which were already calculated, returning the Fibonacci number for $K \leq J - 1$ and default value $D$ otherwise. For a population size of 2000 programs, in generation 22, the program given in table 6.6 emerged. A re-implementation in Lisp together with a simplified version is given in appendix C1. Function *SRF* realizes recursion in an indirect way: each new element in the sequence is defined recursively in terms of one or more previous elements.

Other treatments of recursion in genetic programming can be found for example in Olsson (1995) and Yu and Clark (1998).

### 3.2.3 EVALUATION OF GENETIC PROGRAMMING

Genetic programming is a model-driven approach to learning (see sect. 3.1.1), that is, it starts with a set of programs (hypotheses), these are checked against the data and modified accordingly. Because the method relies on (nearly) blind search, the time effort for coming up with a program solving the desired task is very high. Program construction by generate-and-test has next to nothing in common with the way in which a human programmer develops program code which (hopefully) is guided by the given (complete or incomplete) specification.

Furthermore, effort and success of program construction are heavily dependent on the given representation of the problem and of the pre-defined set of functions and terminals. For the block-stacking example given above, knowledge about which information in a constellation of blocks is relevant for choosing an action was explicitly coded in the sensors. In chapter 8, we will show that our own approach allows to *infer* which information (predicates) are relevant for solving a problem.

*Table 6.7.* Learning the *daughter* Relation

Training Examples:
$\mathcal{E}^+ = \{e_1 = daughter(mary, ann), e_2 = daughter(eve, tom)\}$
$\mathcal{E}^- = \{daughter(tom, ann), daughter(eve, ann)\}$
Background Knowledge:
$\mathcal{B} = \{parent(ann, mary), parent(ann, tom), parent(tom, eve), parent(tom, ian),$
$female(ann), female(mary), female(eve)\}$
To be learned clause: *(learning methods described below)*
$daughter(X, Y) \leftarrow female(X), parent(Y, X)$

## 3.3 INDUCTIVE LOGIC PROGRAMMING

Inductive logic programming (ILP) was named by Muggleton (1991) because this area of research combines inductive learning and logic programming. Most work done in this area concerns concept learning, that is, induction of non-recursive classification rules (see sect. 3.1.1). In principle, all ILP approaches can be applied to learning recursive clauses, if it is allowed that the name of the to be learned relation can be used when constructing the body which characterizes the to be learned relation. But without special heuristics, neither termination of the learning process nor termination of the learned clauses can be guaranteed. In the following, we present basic concepts of ILP, three kinds of learning mechanisms, and biases used in ILP. Finally, we refer some ILP systems which address learning of recursive clauses. An introduction to ILP is given for example by Lavrač and Džeroski (1994) and Muggleton and De Raedt (1994), an overview of program synthesis and ILP is given by Flener (Flener, 1995; Flener and Yilmaz, 1999).

In the following, we give illustrations of the basic concepts and learning algorithms using a simple classification problem presented, for example, in Lavrač and Džeroski (1994). The to be learned concept is the relation *daughter(X,Y)* (see tab. 6.7).

### 3.3.1 BASIC CONCEPTS

In ILP, examples and hypotheses are represented as subsets of first order logic such as definite Horn clauses with some additional restrictions (see discussion of biases below). Definite Horn clauses have the form $T \vee \neg L_1 \vee \ldots \vee \neg L_n$, or in Prolog notation $T \leftarrow L_1, \ldots, L_n$. The positive literal $T$ is called head of the clause and represents the to be learned relation. Often, a clause is represented as a set of literals.

Examples typically can be divided into positive and negative examples. In addition to the examples, background knowledge can be provided (see sect. 3.1.1, table 6.2). The goal of induction is to obtain a hypothesis which,

together with the background knowledge, is complete and consistent with respect to the examples:

**Definition 6.15 (Complete and Consistent Hypotheses)** *Given a set of training examples $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$, with $\mathcal{E}^+$ as positive examples and $\mathcal{E}^-$ as negative examples, together with background knowledge $\mathcal{B}$ and a hypothesis $\mathcal{H}$, let $covers(\mathcal{B} \cup \mathcal{H}, \mathcal{E})$ be a function which returns all elements of $\mathcal{E}$) which follow from $\mathcal{B} \cup \mathcal{H}$. The following conditions must hold:*

**Completeness:** $covers(\mathcal{B} \cup \mathcal{H}, \mathcal{E}^+) = \mathcal{E}^+$.
*Each positive example in $\mathcal{E}^+$ is covered by hypothesis $\mathcal{H}$ together with the background knowledge $\mathcal{B}$. In other words, the positive examples follow from hypothesis and background knowledge ($\mathcal{B} \wedge \mathcal{H} \models \mathcal{E}^+$).*

**Consistency:** $covers(\mathcal{B} \cup \mathcal{H}, \mathcal{E}^-) = \emptyset$.
*No negative example in $\mathcal{E}^-$ is covered by hypothesis $\mathcal{H}$ together with the background knowledge $\mathcal{B}$. In other words, no negative example follows from hypothesis and background knowledge ($\mathcal{B} \wedge \mathcal{H} \wedge \mathcal{E}^- \models \emptyset$).*

In logic programming (Sterling and Shapiro, 1986), SLD-resolution can be used to check for each example in $\mathcal{E}$ whether it is entailed by the hypothesis together with the background knowledge. An additional requirement is, that the positive examples do not already follow from the background knowledge. In this case, learning (construction of a hypothesis) is not necessary ("prior necessity" Muggleton and De Raedt, 1994).

All possible hypotheses – that is, all definite horn clauses with the to be learned relation as head – which can be constructed from the examples and the background knowledge constitute the so called *hypothesis space* or search space for induction. For a systematic search of the hypothesis space, it is useful to introduce a partial order over clauses, based on $\theta$-subsumption (Plotkin, 1969):

**Definition 6.16 ($\theta$-Subsumption)** *Let $c$ and $c'$ be two program clauses. Clause $c$ $\theta$-subsumes clause $c'$, if there exists a substitution $\theta$, such that $c\theta \subseteq c'$. Two clauses $c$ and $d$ are* $\theta$-subsumption equivalent *if $c$ $\theta$-subsumes $d$ and if $d$ $\theta$-subsumes $c$. A clause is* reduced *if it is not $\theta$-subsumption equivalent to any proper subset of itself.*

An example is given in figure 6.5.

Based on $\theta$-subsumption, we can introduce a syntactic notion of generality: If $c\theta \subseteq c'$, then $c$ is at least as general as $c'$, written $c \leq c'$. If $c \leq c'$ holds and $c' \leq c$ does not hold, $c$ is a *generalization* of $c'$ and $c'$ is a *specialization* (refinement) of $c$. The relation $\leq$ introduces a *lattice* on the set of reduced clauses. That is, any two clauses have a least upper bound and a greatest lower bound which are unique except for variable renaming (see figure 6.6).

$c = daughter(X, Y) \leftarrow parent(Y, X), parent(W, V)$
$d = daughter(X, Y) \leftarrow parent(Y, X)$

$c$ and $d$ are $\theta$-subsumption equivalent:
$c\theta \subseteq d$ and $d\theta \subseteq c$ with $\theta = \{W \leftarrow Y, V \leftarrow X\}$

$c$ is not reduced ($d$ is a proper subset of $c$).
$d$ is reduced.

*Figure 6.5.* $\theta$-Subsumption Equivalence and Reduced Clauses

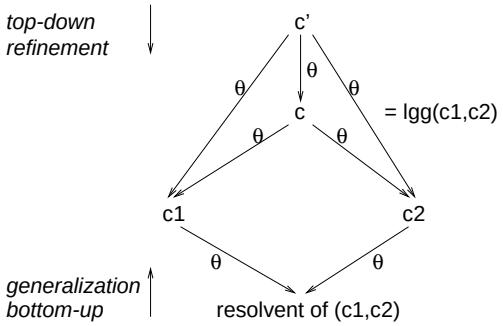



*Figure 6.6.* $\theta$-Subsumption Lattice

Based on $\theta$-subsumption, the least general generalization of two clauses can be defined (Plotkin, 1969)[9]:

**Definition 6.17 (Least General Generalization)** *The least general generalization (lgg) of two reduced clauses $c$ and $c'$, written $lgg(c, c')$ is the least upper bound of $c$ and $c'$ in the $\theta$-subsumption lattice.*

Note, that calculating the resolvent of two clauses is the inverse process to calculating an *lgg*: When calculating an *lgg*, two clauses are generalized by keeping their $\theta$-substitution equivalent common subset of literals. When calculating the resolvent, two clauses are integrated into one, more special clause where variables occuring in the given clauses might be replaced by terms (see description of resolution in sect. 2.2 in chap. 2 and above in this chapter). With $\theta$-subsumption as the central concept for ILP we have the basis for characterizing ILP learning techniques as either bottom-up construction of least general generalizations or top-down search for refinements (see fig. 6.6).

[9]We will come back to this concept applied to *terms* (instead of logical formulae) under the name of first-order anti-unification in chapter 7 and in part III.

*Table 6.8.* Calculating an *rlgg*

*rlgg(daughter(mary,ann),daughter(eve,tom)) =*
*lgg(daughter(mary,ann)*$\leftarrow \mathcal{B}$*, daughter(eve,tom)* $\leftarrow \mathcal{B}$*) =*
*daughter(X,Y)* $\leftarrow$ *female(X), parent(Y,X)*

### 3.3.2 BASIC LEARNING TECHNIQUES

In the following we present two techniques for bottom-up generalization – calculating relative *lgg*s and inverse resolution – and one technique for top-down search in a refinement graph.

**Relative Least General Generalization.** Relative least general generalization is an extension of *lgg*s with respect to background knowledge, for example used in *Golem* (Muggleton and Feng, 1990).

**Definition 6.18 (Relative Least General Generalization)** *The relative least general generalization (rlgg) of two clauses* $c_1$ *and* $c_2$ *is their least general generalization* $lgg(c_1, c_2)$ *relative to the background knowledge* $\mathcal{B}$*:* $rlgg(c_1, c_2) = lgg(c_1 \leftarrow \mathcal{B}, c_2 \leftarrow \mathcal{B})$.

In the ILP literature, enriching clauses with background knowledge is discussed under the name of *saturation* (Rouveirol, 1991).

An algorithm for calculating *lgg*s for clauses is, for example, given in Plotkin (1969). In table 6.8 we demonstrate calculating *rlgg*s with the *daughter* example (tab. 6.7). The *rlgg* of the two positive examples is calculated by determining the *lgg* between Horn clauses where the examples are consequences of the background knowledge.

**Inverse Resolution.** Inverse resolution is a generalization technique based on inverting the resolution rule of deductive inference. Inverse resolution requires inverse substitution:

**Definition 6.19 (Inverse substitution)** *For a logical formula* $W$*, an inverse substitution* $\theta^{-1}$ *of a substitution* $\theta$ *is a function that maps terms in* $W\theta$ *to variables, such that* $W\theta\theta^{-1} = W$.

Again, we do not present the algorithm, but illustrate inverse resolution with the *daughter*-example introduced in table 6.7 (see fig. 6.7). We restrict our positive examples to $e_1$ = *daughter(mary, ann)* and background knowledge to $b_1$ = *female(mary)* and $b_2$ =*parent(ann, mary)*. Working with bottom-up generalization, the initial hypothesis is given as the first positive example. The learning process starts, for example, with clauses $e_1$ and $b_2$. Inverse resolution attempts to find a clause $c_1$ such that $c_1$ together with $b_2$ entails $e_1$. If such

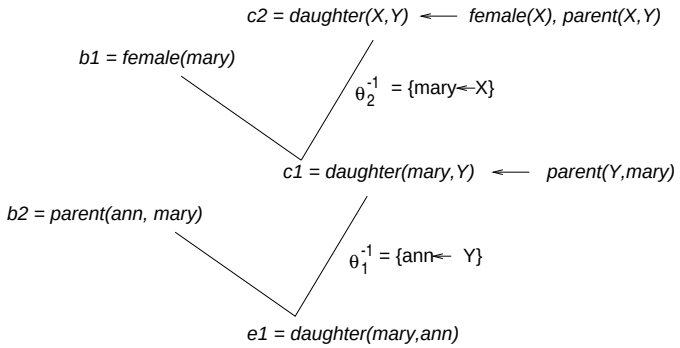



*Figure 6.7.* An Inverse Linear Derivation Tree (Lavrač and Džeroski, 1994, pp. 46)

a clause can be found, it becomes the current hypothesis and a further clause from the set of positive examples or the background knowledge is considered. The current hypothesis is attempted to be inversely resolved with this new clause and so on. The inverse resolution operator used in the example is called *absorption* or V operator.

In general, inverse resolution involves backtracking. In systems working with inverse resolution, it is sometimes necessary to introduce new predicates in the hypothesis for constructing a complete and consistent hypothesis. The operator realizing such a *necessary predicate invention* is called W operator (Muggleton, 1994). Inverse resolution is for example used in *Marvin* (Sammut and Banerji, 1986).

**Search in the Refinement-Graph.** Alternatively to bottom-up generalization, hypotheses can be constructed by top-down refinement. The step from a general to a more specific hypothesis is realized by a so called refinement operator:

**Definition 6.20 (Refinement Operator)** *Given a hypothesis language $\mathcal{L}$ (e. g., definite Horn clauses), a refinement operator $\rho$ maps a clause $c$ to a set of clauses $\rho(c) = \{c' \mid c' \in \mathcal{L}, c < c'\}$.*

Typically, a refinement operator computes so called "most general specifications" under $\theta$-subsumption by performing one of the following syntactic operations:

- apply a substitution to $c$,
- add a literal to the body of $c$.

*Table 6.9.* Simplified MIS-Algorithm (Lavrač and Džeroski, 1994, pp.54)

Initialize hypothesis $\mathcal{H}$ to a (possibly empty) set of clauses in $\mathcal{L}$.

**REPEAT**

Read next (positive or negative) example.

**REPEAT**

**IF** there exists a covered negative example *e*
**THEN** delete incorrect clauses from $\mathcal{H}$.

**IF** there exists a positive example *e* covered by $\mathcal{H}$
**THEN** with *breadth-first search* of the refinement graph, develop a clause *c* which covers *e* and add it to $\mathcal{H}$.

**UNTIL** $\mathcal{H}$ is complete and consistent.

**Output:** Hypothesis $\mathcal{H}$.

**FOREVER**.

Learning as search in refinement graphs was first introduced by Shapiro (1983) in the interactive system *MIS* (Model Inference System). An outline of the *MIS*-algorithm is given in table 6.9.

Again, we illustrate the algorithm with the *daughter* example (tab. 6.7). A part of the breadth-first search tree (see chap. 2) is given in figure 6.8. The root node of an refinement graph is formally the most general clause, that is *false* or the empty clause. Typically, a refinement algorithm starts with the most general definition of the goal relation, for our example, that is $\mathcal{H} = c =$ *daughter(X,Y)* $\leftarrow$ *true*. The hypothesis covers both positive training examples, that is, it must not be modified if these examples are presented initially. If a negative example (e. g., $e_3 =$ *daughter(eve, tom)*) is presented, the hypothesis needs to be modified because $c$ covers the negative examples, too. Therefore, a clause covering the positive examples but not covering $e_3$ must be constructed. The set of all possible (minimal) specializations of $c$ is calculated as $\rho(c) = \{daughter(X, Y) \leftarrow L\}$. The newly introduced literal is either a literal which has the variables used in the clausal head as argument or a literal introducing a new variable. The predicate names of the literals can be obtained by the background knowledge, additional build-in predicates (such as equality or inequality) can be used.

Already for this small example, the search space becomes rather large: $L$ can be

- $X = Y$, or $female(X)$, or $female(Y)$, or $parent(X, X)$, or $parent(X, Y)$, or $parent(Y, X)$, or $parent(Y, Y)$, or

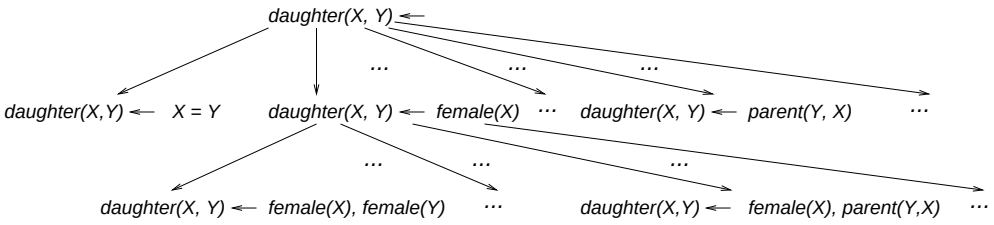



*Figure 6.8.* Part of a Refinement Graph (Lavrač and Džeroski, 1994, p. 56)

- $parent(X, Z)$, or $parent(Z, X)$, or $parent(Y, Z)$, or $parent(Z, Y)$, with $Z \neq X$ and $Z \neq Y$.

Note, that in this example the language is restricted such, that literals with the goal predicate *daughter* are not considered in the body of the clause and therefore, no recursive clauses are generated.

The refinement $c' = daughter(X, Y) \leftarrow female(X)$ covers the positive examples and does not cover $e_3$. Therefore, it becomes the new hypothesis. If another negative example, such as $e_4$ = *daughter(eve, ann)* is presented, the hypothesis again must be modified, because $c'$ covers this example, and so on.

A current system, based on top-down refinement is *Foil* (Quinlan, 1990).

### 3.3.3 DECLARATIVE BIASES

The most general setting for learning hypotheses in an ILP framework would be to allow that the hypothesis can be represented as an arbitrary set of definite Horn clauses, that is any legal Prolog program. Because ILP techniques are heavily dependent on search in hypothesis space, typically much more restricted hypothesis languages are considered. In the following, we present such language restrictions, also called syntactic biases (see sect. 3.1.1). Additionally, semantic biases can be introduced to restrict the to be learned relations.

**Syntactic Biases.** One possibility to restrict the search space is, to provide second order schemes which represent the form of searched for hypotheses (Flener, 1995). A *second order scheme* is a clause with existentially quantified predicate variables. An example scheme and an instantiation of such a scheme are (Muggleton and De Raedt, 1994, p. 656):

**Scheme:** $\exists p, q, r : p(X, Y) \leftarrow q(X, XW), q(YW, Y), r(XW, YW)$.

**Instantiation:** *connected(X,Y)* ← *partof(X,XW), partof(Y,YW), touches(XW,YW).*

A typical syntactic bias is to restrict search to linked clauses:

*mode(1, append(+list,+list,-list))*
*mode(*, append(-list,-list,+list))*
*list(nil)* ←
*list([X|T])* ← *integer(X), list(T)*

*Figure 6.9.* Specifying Modes and Types for Predicates

**Definition 6.21 (Linked Clause)** *A clause is linked if all of its variables are linked. A variable $V$ is linked in a clause $c$ iff $V$ occurs in the head of $c$, or there exists a literal $l$ in $c$ that contains variables $V$ and $W$, $W \neq V$ and $W$ is linked in $c$.*

The scheme given above represents a linked clause.

Many ILP systems provide a number of parameters which can be set by the user. These parameters present biases and by assigning them with different values, so called *bias shifts* can be realized.

Two examples of such parameters are the depth of terms and the level of terms.

**Definition 6.22 (Depth of a Term)** *The depth $d(V)$ of a variable is $0$. The depth $d(c)$ of a constant is $1$. The depth $d(f(t_1, \ldots, t_n))$ of a term is $1 + max(\{d(t_i)\})$.*

**Definition 6.23 (Level of a Term)** *The level $t(t)$ of a term $t$ in a linked clause $c$ is $0$ if $t$ occurs as an argument in the head of $c$, and $1 + min(\{l(s)\})$ where $s$ and $t$ occur as arguments in the same literal of $c$.*

In the scheme given above, $X$ and $Y$ have depth 1 and $XW$ and $YW$ have depth 2. For linked languages with maximal depth 1 and $level > 1$, the number of literals in the body of the clause can grow exponentially with its level (Muggleton and De Raedt, 1994).

**Semantic Biases.** A typically semantic bias is to provide information about the modes and types of a to be learned clause. This information can be provided by specifying declarations of predicates in the background knowledge. An example for specifying modes and types for *append* is given in figure 6.9: If the first two arguments of *append* are instantiated (indicated by $+list$), this predicate succeeds once (1),; if the last argument is instantiated, it can succeed finitely many times ($*$). The predicate is restricted to lists of integers. It is obvious that such a semantic bias can restrict effort of search considerably.

Another semantic bias, for example used in *Foil* (Quinlan, 1990) and *Golem* (Muggleton and Feng, 1990), is the notion of determinate clauses.

**Definition 6.24 (Determinate Clauses)** *A definite clause $h \leftarrow l_1, \ldots, l_n$ is determinate (with respect to background knowledge $\mathcal{B}$ and examples $\mathcal{E}$) iff for every substitution $\theta$ for $h$ that unifies $h$ to a ground instance $e \in \mathcal{E}$, and for all $i = 1, \ldots n$ there is a unique substitution $\theta_i$ such that $(l_1 \wedge \ldots \wedge l_i)\theta\theta_i$ is ground and is true for $\mathcal{B} \wedge \mathcal{E}^+$.*

A clause is determinate if all its literals are determinate and a literal $l_i$ is determinate if each of its variables which do not appear in preceding literals $l_j$, $j = 1, \ldots i-1$ has only one possible instantiation given the instantiations in $\{l_j \mid j = 1, \ldots i-1\}$ . For the daughter example given in table 6.7, the clause $daughter(X, Y) \leftarrow female(X), parent(X, Y)$ is determinate: If $X$ is instantiated with *mary*, then $Y$ can only be instantiated with *ann*.

Restriction to determinate clauses can reduce the exponential growth of literals in the body of a hypotheses during search. A special case of determinacy is the so called $ij$-determinacy. Parameter $i$ represents the maximum depth of variables (def. 6.22), parameter $j$ represents the *maximal degree* of dependency (def. 6.23) in determined clauses (Muggleton and Feng, 1990).

**Definition 6.25 ($ij$-Determinacy of Clauses)** *A clause consisting of a head only is $0j$-determinate. A clause $h \leftarrow l_1, \ldots, l_m, l_{m+1}, \ldots, l_n$ is $ij$-determinate iff*

- *$h \leftarrow l_1, \ldots, l_m$ is $(i-1)j$-determinate, and*
- *every literal in $l_{m+1}, \ldots, l_n$ contains only determinate terms and has a degree at most $j$.*

The clause $daughter(X, Y) \leftarrow female(X), parent(X, Y)$ is 11-determinate because all variables in the body appear in the head ($i = 1$) and the maximal degree is one ($j = 1$) for variable $Y$ in *parent(X,Y)* which depends on $X$.

It can be shown that the restriction to $ij$-determinate clauses allows for polynomial effort of learning (Muggleton and Feng, 1990).

### 3.3.4 LEARNING RECURSIVE PROLOG PROGRAMS

In principle, all ILP systems can be extended to learning recursive clauses, that is, they can be seen as program synthesis systems, if the language bias is relaxed such that the goal predicate is allowed to be used when constructing the body of a hypothesis. For the systems *MIS* (Shapiro, 1983), *Foil* (Quinlan, 1990), and *Golem* (Muggleton and Feng, 1990), for example, application to program synthesis was demonstrated. In the following, we first show, how learning recursive clauses is realized with *Golem* and afterwards introduce some ILP approaches which were proposed explicitly for program synthesis.

*Table 6.10.* Background Knowledge and Examples for Learning *reverse(X,Y)*

```
%% positive examples
rev([],[]). rev([1],[1]). rev([2],[2]). rev([3],[3]). rev([4],[4]).
rev([1,2],[2,1]). rev([1,3],[3,1]). rev([1,4],[4,1]). rev([2,2],[2,2]).
rev([2,3],[3,2]). rev([2,4],[4,2]). rev([0,1,2],[2,1,0]).
rev([1,2,3],[3,2,1]).
%% negative examples
rev([1],[]). rev([0,1],[0,1]). rev([0,1,2],[2,0,1]).
app([1],[0],[0,1]).
%% background knowledge
!- mode(rev(+,-)).
!- mode(app(+,+,-)).
rev([],[]). ... rev([1,2,3],[3,2,1]). %% all positive examples
app([],[],[]). app([1],[],[1]). app([2],[],[2]). app([3],[],[3]).
app([4],[],[4]). app([],[1],[1]). app([],[2],[2]). app([],[3],[3]).
app([],[4],[4]). app([1],[0],[1,0]). app([2],[1],[2,1]).
app([3],[1],[3,1]). app([4],[1],[4,1]). app([2],[2],[2,2]).
app([3],[2],[3,2]). app([4],[2],[4,2]). app([2,1],[0],[2,1,0]).
app([3,2],[1],[3,2,1]).
```

**Learning *reverse* with *Golem*.** The system *Golem* (Muggleton and Feng, 1990) works with bottom-up generalization, calculating relative least general generalizations (see def. 6.18). We give an example, how *reverse(X, Y)* can be induced with *Golem*. In table 6.10 the necessary background knowledge, positive and negative examples are presented.

As described above, an *rlgg* is calculated between pairs of clauses of the form $e \leftarrow \mathcal{B}$ where $e$ is a positive example and $\mathcal{B}$ is the conjunction of clauses from the background knowledge. The "trick" used for learning recursive clauses with an *rlgg* technique is that the positive examples are additionally made part of the background knowledge. In this way, the name of the goal predicate can be introduced in the body of the hypothesis. The notion of $ij$-determinacy (see def. 6.25) is used to avoid exponential size explosion of *rlggs*.

For the first examples *rev([ ],[ ])* and *rev([1],[1])* the *rlgg* is:

$$\begin{aligned} rev(X,X) \leftarrow \quad & rev([X],[X]), rev([2],[2]), \ldots, \\ & rev([X,2],[2,X]), \ldots, rev([X,X,2],[2,X,X]), \\ & app(X,X,X), app([X],[],[X]), \\ & app([2],[],[2]), \ldots, app([3,2],[X],[3,2,X]). \end{aligned}$$

This hypothesis has a head which is still too special and a body which contains far too many literals. The head will become more general in the learning process when further positive examples are introduced. Literals from the body can be reduced by eliminating literals which cover negative examples. In general, this can be a very time consuming task because all subsets of the

literals in the body must be checked. In Muggleton and Feng (1990) a technique working with $O(n^2)$ is described.

For the above example, it remains:

$$rev(X, X) \leftarrow \quad app(X, X, Y).$$

The finally resulting program is:

$$\begin{aligned} rev([A|B], [C|D]) \leftarrow \quad & reverse(B, E), \\ & append(E, [A], [C|D]). \end{aligned}$$

A trace generated by *Golem* for this example is given in appendix C2.

Note, that no base case is learned, that is, interpretation of *rev* would not terminate. Furthermore, the sequence of literals in a clause depends on the sequence of literals in the background knowledge. If in our example, the *append* literals would be presented before the *rev* literals, the resulting program would have a body where *append* appears before *rev*. In using a top-down left-to-right strategy, such as Prolog, interpretation of the clause would fail.

Instead of presenting the *append* predicate necessary for inducing *reverse* by ground instances, a predefined *append* function can be presented as background knowledge. But this can make induction harder because for calculating an *rlgg* the background knowledge must be grounded. That is, a recursive definition in the background knowledge must be instantiated and unfolded to some fixed depth. Because there might be many possible instantiations and because the "necessary" depth of unfolding is unknown, search effort can easily explode. Without a predefined restriction of unfolding-depth, termination cannot be guaranteed.

**Special Purpose ILP Synthesis Systems.** ILP systems explicitly designed for learning recursive clauses are for example *TIM* (Idestam-Almquist, 1995), *Synapse* (Flener, Popelinsky, and Stepankova, 1994), and *Merlin* (Boström, 1996). The hypothesis language of *Tim* consists of tail recursive clauses together with base clauses. Learning is performed again by calculating *rlgg*s. *Synapse* synthesizes programs which confirm to a divide-and-conquer scheme and it incorporates predicate invention. *Merlin* learns tail recursive clauses based on a grammar inference technique (see sect. 3.1.2). For a given set of positive and negative examples, a deterministic finite automaton is constructed which accepts all positive and no negative examples. This can for example be realized using the ID algorithm of Angluin (1981). The automaton is then transformed in a set of Prolog rules. *Merlin* also uses predicate invention.

## 3.4 INDUCTIVE FUNCTIONAL PROGRAMMING

In contrast to the inductive approaches presented so far, the classical functional approach to inductive program synthesis is based on a two step process:

**Rewriting I/O Examples into constructive expressions:** Example computations, presented as input/output examples are transformed into traces and predicates. Each predicate characterizes the structure of one input example. Each trace calculates the corresponding output for a given input example.

**Recurrence Detection:** Regularities are searched for in the set of predicates/traces pairs. These regularities are used for generating a recursive generalization.

The different approaches typically use a simple strategy for the first step which is not discussed in detail. The critical differences between the functional synthesis algorithms is how the second step is realized. Two pioneering approaches are from Summers (1977) and Biermann (1978). The work of Summers was extended by several researchers, for example by Jouannaud and Kodratoff (1979), Wysotzki (1983), and Le Blanc (1994).

In the following, we first present Summers'approach in some detail, because our own work is based on his approach. Afterwards we give a short overview over Biermann's approach and finally we discuss some extensions of Summers' work.

### 3.4.1 SUMMERS' RECURRENCE RELATION DETECTION APPROACH

**Basic Concepts.** Input to the synthesis algorithm (called *Thesys*) is a set of input/output examples $E = \{e_1, \ldots, e_k$ with $e_j = i_j \rightarrow o_j$, $j = 1, \ldots, k$. The to be constructed output is a recursive Lisp function which generalizes over $E$. The following Lisp primitives[10] are used:

- Atom `nil`: representing the empty list or the truth-value *false*.
- Predicate `atom(x)`: which is true if $x$ is an atom.
- Constructor `cons(x, l)`: which inserts an element $x$ in front of $l$.
- Basic functions: selectors `car(x)` and `cdr(x)`, which return the first element of a list/a list without its first element;
  and compositions of selectors, which can be noted for short as $c\langle x\rangle^+r$, $x \in \{a, d\}$ for instance *car(cdr(x))* = *cadr(x)*.

[10] For better readability, we write the operator names, followed by comma-separated arguments in parenthesis, instead of the usual Lisp notation. That is, for the Lisp expression `(cons x l)`, we write *cons(x, l)*.

- McCarthy-Conditional `cond(b1, ..., bn)`: where $b_i$ are pairs of predicates and operations $p_i(x) \rightarrow f_i(x)$.

Inputs and outputs of the searched for recursive function are represented as single s-expressions (simple expressions):

**Definition 6.26 (S-Expression)** *Atom* `nil` *and basic functions are s-expressions.* `cons(s1, s2)` *is an s-expression, if* `s1` *and* `s2` *are s-expressions.*

With Summers' method, Lisp functions with the following underlying *basic program scheme* can be induced:

**Definition 6.27 (Basic Program Scheme)**

$$\begin{aligned} F(x) \leftarrow \quad & (p_1(x) \rightarrow f_1(x), \\ & \ldots, \\ & p_k(x) \rightarrow f_k(x), \\ & T \rightarrow C(F(b(x)), x)) \end{aligned}$$

*where:*
$p_1, \ldots p_k$ *are predicates of the form* $atom(b_i(x))$
$b, b_i$ *are basic functions*
$f_1, \ldots, f_k$ *are s-expressions,*
$T$ *is the truth-value "true",* $C(w, x)$ *is a cons-expression with* $w$ *occuring exactly once.*

This basic program scheme allows for arbitrary *linear* recursion, that is, more complex forms, as tree recursion are out of reach of this method. For illustration we give an example presented in Summers (1977) in figure 6.10. For better readability, we represent inputs and outputs as lists and not as s-expressions. In the following, we will describe, how induction of a recursive function is realized by (1) generating traces, and (2) detecting regularities (recurrence) in traces.

**Generating Example Traces.** In the first step of synthesis, the input/output examples are transformed into a non-recursive program which covers exactly the given examples. This transformation involves:

- constructing a trace $f_j$ which calculates $o_j$ for $i_j$ for all examples $e_j$, $j = 1, \ldots, k$,

- determining a predicate which associates each input with the trace calculating the desired output, and

Input/Output Examples:
$\{nil \rightarrow nil,$
$(A) \rightarrow ((A)),$
$(A\ B) \rightarrow ((A)\ (B)),$
$(A\ B\ C) \rightarrow ((A)\ (B)\ (C))\}$

Recursive Generalization:

$$\begin{array}{ll} unpack(x) \leftarrow & (atom(x) \rightarrow nil, \\ & T \rightarrow u(x)) \\ u(x) \leftarrow & (atom(cdr(x)) \rightarrow cons(x, nil), \\ & T \rightarrow cons(cons(car(x), nil), u(cdr(x)))) \end{array}$$

*Figure 6.10.* Learning Function *unpack* from Examples

- constructing a program with the following underlying scheme:

$$\begin{array}{ll} F(x) \leftarrow & (p_1(x) \rightarrow f_1(x), \\ & \ldots, \\ & p_k(x) \rightarrow f_k(x)). \end{array}$$

A trace is a function which calculates an output for a given input. In Summers' approach, such functions are always *cons*-expressions. That such a *cons*-expression does exist, all atoms appearing in the output, must also appear in the input. That is, it is not possible, to generate new symbols in an output. That such a *cons*-expression is uniquely determined, each atom appearing in the output, must appear exactly once in the input. The reason for that restriction is, as we will see below, that the output is constructed from the *syntactical structure* of the input and therefore, the position of each used atom must be unique.

The algorithm for constructing the traces is given in table 6.11.[11] An example is given in figure 6.11.

In the next step, predicates must be determined, which characterize the example inputs in such a way that each input can be associated with its correct trace: For the set of predicates must hold that $p_i(x_i)$, $i = 1, \ldots, k$ evaluate to $T$ (*true*) and that predicates $p_j(x_i)$, $1 \leq j \leq i$ evaluats to *nil*. Because the only predicate used is *atom*, the inputs are discriminated by their structure only. For catching characteristics of the atoms themselves, "semantic" predicates, such as *equal* or *greater* would be needed.

The searched for predicates have the form *atom($b_i$, x)* where $b_i$ is a basic function. The algorithm for determining the predicates must construct such

[11] $ST$ stands for "semi-trace". A semi-trace is a trace where the sequence of evaluation of expressions is not fixed.

*Table 6.11.* Constructing Traces from I/O Examples

Let *SE* be the set of expressions over basic functions $b$, which describe sub-expressions $sub(x)$ of an s-expression $x$, associated with that function, that is $SE = \{(b(x), sub(x))\}$.
Let $x$ be an example input and $y$ be the associated output.

$$ST(x,y) = \begin{cases} b(x) & y \neq nil \; and \; (b(x), y) \in SE \\ cons(ST(x, car(y)), ST(x, cdr(y))) & otherwise \end{cases}$$

---

Sub-expressions of $i = (A\ B)$ can be described by
$SE = \{(i, (A\,B)), (car(i), A), (cdr(i), (B)), (cadr(i), B), (cddr(i), nil)\}$

For the given examples, the traces constructed with $ST$ are:

| | |
|---|---|
| $nil \rightarrow nil$: | nil |
| $(A) \rightarrow ((A))$: | cons(x, nil) |
| $(A\ B) \rightarrow ((A)\ (B))$: | cons(cons(car(x), nil), cons(cdr(x), nil)) |
| $(A\ B\ C) \rightarrow ((A)\ (B)\ (C))$: | cons(cons(car(x), nil), cons(cons(cadr(x),nil),cons(cddr(x),nil))) |

---

*Figure 6.11.* Traces for the *unpack* Example

*Table 6.12.* Calculating the Form of an S-Expression

List $(A\ (A\ B)\ C)$ is represented by
S-Expression $(A.(B.(C.nil)).(D.nil))$.
$form((A.(B.(C.nil)).(D.nil))) = (\omega.(\omega.(\omega.\omega)).(\omega.\omega))$

basic-functions that, if $x_i < x_{i+1}$, $b_i(x_i)$ is an atom, but $b_i(x_{i+1})$ is not. To decide whether an input is smaller than an other input, we need an ordering relation over possible inputs (that is over s-expressions). In a first step, we reduce the inputs to their structure, by replacing all atoms by identical atoms $\omega$:

**Definition 6.28 (Form of an S-Expression)**

$$form(x) = \begin{cases} \omega & atom(x) = T \\ cons(form(car(x)), form(cdr(x))) & otherwise. \end{cases}$$

An example is given in table 6.12.
Now we can define an order relation over the set of forms s-expressions:

$$
\begin{aligned}
F_L(x) \leftarrow \quad & (atom(x) \rightarrow nil,\\
& atom(cdr(x)) \rightarrow cons(x, nil),\\
& atom(cddr(x)) \rightarrow cons(cons(car(x), nil), cons(cdr(x), nil)),\\
& T \rightarrow cons(cons(car(x), nil), cons(cons(cadr(x), nil), cons(cddr(x), nil))))
\end{aligned}
$$

*Figure 6.12.* Result of the First Synthesis Step for *unpack*

**Definition 6.29 (Complete Partial Order over S-Expressions)** *The complete partial order over $SF$ (forms of s-expressions) is recursively defined as:*
$\forall s \in SF : \omega \leq s$
$\forall a, b, c, d \in SF : (a.b) \leq (c.d) \Leftrightarrow (a \leq c) \wedge (b \leq d)$.

*The complete partial order over s-expressions $S$ is:*
$\forall s, t \in S : s \leq t \Leftrightarrow form(s) \leq form(t)$.

Note, that the order is partial, because not each s-expression can be compared with each other. For example, it holds neither that *(A (B C)) ≤ ((A B) C)* nor that *((A B) C) ≤ (A (B C))*. For constructing predicates, it must hold that the example inputs can be totally ordered. For the synthesized function it is assumed that its hypothetical (infinite) input domain is totally ordered, too.

The algorithm for constructing predicates is defined as follows:

**Definition 6.30 (Constructing Predicates)** $PredGen(x_i, x_{i+1}) = PG(x_i, x_{i+1}, I)$, *where $I$ is the identity function*

$$
PG(x, y, \theta) = \begin{cases} \emptyset & atom(y) \\ \{\theta\} & atom(x)\ and\ \neg atom(y) \\ PG(car(x), car(y), car \circ \theta) \cup & \\ PG(cdr(x), cdr(y), cdr \circ \theta) & otherwise. \end{cases}
$$

For example, *PredGen((A B), (A B C))* returns $\{cddr\}$ and the resulting predicate is *atom(cddr(x))*.

The resulting non-recursive function which covers the set of input examples is given in figure 6.12.

**Recurrence Detection.** The second step of synthesis is to generalize over the given set of examples. This can be done with a purely syntactical approach – by pattern matching: The finite program composed from traces is checked for regularities between pairs of traces. This is done by determining differences between $f_i(x)$ and $f_{i+c}(x)$ and checking whether recurrence relation $f_{i+c} = C(f_i(b(x)), x)$ holds for all examples. The differences and the resulting recurrence relation for the *unpack* example is given in figure 6.13.

In Summers (1977) no algorithm for calculating the regularities is presented, except that he mentions that the algorithm is similar to unification algorithms.

Differences:
$f_2(x) = cons(x, f_1(x))$
$f_3(x) = cons(cons(car(x), nil), f_2(cdr(x)))$
$f_4(x) = cons(cons(car(x), nil), f_3(cdr(x)))$

$p_2(x) = p_1(cdr(x))$
$p_3(x) = p_2(cdr(x))$
$p_4(x) = p_3(cdr(x))$
Recurrence Relation:
$f_1(x) = nil$
$f_2(x) = cons(x, f_1(x))$
$f_{k+1}(x) = cons(cons(car(x), nil), f_k(cdr(x)))$ for $k = 2, 3$

$p_1(x) = atom(x)$
$p_{k+1}(x) = p_k(cdr(x))$ for $k = 1, 2, 3$

*Figure 6.13.* Recurrence Relation for *unpack*

A detailed description of a method for recurrence detection in final program terms is presented in chapter 7.

From a recurrence relation which holds for a set of examples for indices $k = s, \ldots, f$ where $s$ and $f$ are constants, it is inductively generalized, that this relation holds for the complete domain $k = s, \ldots, n$ with $n \to \infty$. Summers presented a theoretical foundation for this generalization, in form of a set of synthesis theorems. In the following, we present the central aspects of Summers' theory.

**Synthesis Theorems.**

**Definition 6.31 (Approximation of a Function)** *The $k$-th approximation $F_k(x)$ of a function $\mathbf{F}(x)$ is defined as*

$$\begin{aligned} F_k(x) \leftarrow \quad & (p_1(x) \to f_1(x), \\ & \cdots \\ & p_{k-1}(x) \to f_{k-1}(x), \\ & T \to \Omega) \end{aligned}$$

*where $\Omega$ is undefined.*

That is, the finite function $F(x)$ constructed over a set of examples, which is undefined for all inputs with $p_{k-1}(x) = nil$ for all $k > f$ where $f$ is a constant, can be seen as the $k$-th approximation of the to be induced function $\mathbf{F}(x)$.

**Definition 6.32 (Recurrence Relation)** *If there exists an initial point $j$ and an interval $n$ such that $1 \leq j < k$ in an $k$-th approximation $F_k(x)$ defined by*

*a set of examples such that the following fragment differences exist:*

$$\begin{array}{ll} f_{j+n}(x) = & C(f_j(b_1(x)), x), \\ f_{j+n+1}(x) = & C(f_{j+1}(b_1(x)), x), \\ & \cdots \\ f_{k-1}(x) = & C(f_{k-n-1}(b_1(x)), x), \end{array}$$

*and such that the following predicate differences exist:*

$$\begin{array}{ll} p_{j+n}(x) = & p_j(b_2(x)), \\ p_{j+n+1}(x) = & p_{j+1}(b_2(x)),, \\ & \cdots \\ p_{k-1}(x) = & p_{k-n-1}(b_2(x)), \end{array}$$

*then we define the* functional recurrence relation *for the examples to be* $f_1(x), \ldots, f_j(x), \ldots, f_{j+n-1}(x), f_{i+n}(x) = C(f_i(b_1(x)), x)$ *for* $j \leq i \leq k-n-1$, *and we define the* predicate recurrence relation *for the examples to be* $p_1(x), \ldots, p_j(x), \ldots, p_{j+n-1}(x), p_{i+n}(x) = p_i(b_2(x))$ *for* $j \leq i \leq k-n-1$

The index $j$ gives the first position in the set of ordered traces where the recurrence relation holds. All traces with smaller indices are kept as explicit predicate/function pairs. For the *unpack* example holds $j = 2$. The index $n$ gives a fixed interval for the recurrence covering the fact that, in general, recurrence can occur not only between immediately succeeding traces. For the *unpack* example holds $n = 1$.

**Definition 6.33 (Induction)** *If a functional and a predicate recurrence relation exist for a set of examples such that* $k - j \geq 2n$, *then we* inductively infer *that these relationships hold for all* $i \geq j$.

The condition $k - j \geq 2n$ ensures that at least one regularity (between two traces) can be found in the example. We will see in chapter 7, that when considering more general recursive schemes, at least four traces from which regularities can be extracted are necessary.

Applying the recurrence relation induced from the examples, the set of examples can be inductively extended to further approximations of the searched for function $\mathbf{F}$. The $m$-th approximation, with $m \geq j$, has the form:

$$\begin{array}{ll} F_m(x) \leftarrow & (p_1(x) \rightarrow f_1(x), \\ & \cdots \\ & p_k(x) \rightarrow f_k(x), \\ & p_{k+1}(x) \rightarrow f_{k+1}(x), \\ & \cdots \\ & p_m(x) \rightarrow f_m(x), \\ & T \rightarrow \Omega). \end{array}$$

**Lemma 6.1 (Order of Approximations)**
*The set of approximating functions, if it exists, is a chain with partial order* $\leq_F$*, where* $F(x) \leq_F G(x)$ *holds if it is true that, for all* $x$ *for which* $F(x)$ *is defined,* $F(x) = G(x)$*.*

**Proof 6.3 (Order of Approximations)**
*We define* $D_m = \{x \mid p_i(x)\} \cup \ldots \cup \{x \mid p_m(x)\}$ *to be the domain of the approximating function* $F_m(x)$*. From this it is clear that* $F_i(x) \leq_F F_{i+1}(x)$ *for all* $i$*, since* $D_i$ *is a subset of* $D_{i+1}$ *(Summers, 1977, p. 167).*

Now, the searched for function $\mathbf{F}(x)$, which was specified by examples, can be defined as supremum of a chain of approximations:

**Definition 6.34 (Supremum of a Chain of Approximations)** *If a set of examples defines the chain* $\langle\{F_m(x)\}, \leq_F\rangle$*, the searched for function* $\mathbf{F}(x)$*, defined by the examples, is* $\mathbf{sup}\{F_m(x)\}$ *or the limit of* $F_m(x)$ *for* $m \to \infty$*.*

The concepts introduced, correspond to the concepts used in the fixpoint theory of semantics of recursive functions (see appendix B1 and sect. 2 in chap. 7). Summers' basic synthesis theorem represents the converse of this problem, that is, to find a recursive program from a recurrence relation characterization of a partial function. In other words, the synthesis problem is to induce a *folding* from a set of traces considered as *unfolding* of some unknown recursive function.

**Theorem 6.3 (Basic Synthesis)**
*If a set of examples defines* $\mathbf{F}(x)$ *with recurrence relations* $f_1(x), \ldots, f_n(x)$*,* $f_{i+n}(x) = C(f_i(b(x)), x)$*,* $p_1(x), \ldots, p_n(x), p_{i+n}(x) = p_i(b(x))$ *for* $i \leq 1$*, then* $\mathbf{F}(x)$ *is equivalent to the following recursive program:*

$$\begin{aligned} F(x) \leftarrow \quad & (p_1(x) \to f_1(x), \\ & p_n(x) \to f_n(x), \\ & T \to C(F(b(x)), x)). \end{aligned}$$

*Proof: see Summers (1977, pp. 168).*

This basic synthesis theorem provides the foundations of synthesis of recursive programs from a finite set of examples. But it is based on constraints which only allow for very restricted induction. The first restriction is, that recurrence relationships must hold for all traces – not allowing a finite set of special cases which can be dealt with separately. The second restriction is, that the basic function $b$ used in the predicate and functional recurrences must be identical. Summers (1977) presents extensions of the basic theorem which overcome these restrictions.

Summers points out that it would be desirable to give a formal characterization of the class of recursive functions which can be synthesized using this approach, but that such a formal characterization does not exist. In later work, Le Blanc (1994) tries to give such a characterization. In our own work, we give a structural characterization of the class of recursive programs which can be induced from finite programs (see chapter 7). A tight characterization of the class of (semantic) functions which can be synthesized from traces is still missing!

**Variable Addition.** Summers (1977) presents an extension of his approach for cases, where predicate recurrence relations but no functional recurrence relations can be detected in the traces. This problem was later also discussed by Jouannaud and Kodratoff (1979) and Wysotzki (1983). Neither in Summers' nor in later work, it is pointed out, that this problem occurs exactly in such cases when the underlying recurrence relation corresponds to a simple loop, that is a tail recursion. All authors propose solutions based on the introduction of additional variables. This is a standard method in transforming linear recursion into tail recursion in program optimization (Field and Harrison, 1988).

We illustrate this problem with the *reverse* function given in figure 6.14. For the initial traces, all differences have different forms, that is, no recurrence relation can be derived from the examples. The following heuristics can be used: Search for a subexpression $a$, which is identical in all traces. Rewrite the fragments to $g_i(x, a) = f_i(a)$, abstract to $g(x, y)$, and try again to find recurrence relations.

For the *reverse* example, it holds that $a = nil$. After abstraction, for each pair of traces two differences exist. One of these differences is regular and can be transformed into a recurrence relation. For the $g_i(x, y)$, the recursive function $G(x, y)$ can be induced and the resulting program is: $F(x) = G(x, a)$. Variable $y$ plays the role of a collector, where the resulting output is constructed, starting with the "neutral element" with respect to *cons*, which is *nil*. In the terminating case, the current value of $y$ is returned.

### 3.4.2 EXTENSIONS OF SUMMERS' APPROACH

The most well-known extensions of Summers' approach were presented by Jouannaud, Kodratoff and colleagues (Kodratoff and J.Fargues, 1978; Jouannaud and Kodratoff, 1979; Kodratoff, Franova, and Partridge, 1989). In the work of this group, the recursive programs which can be synthesized from traces correspond to a more complex program scheme than that underlying Summers':

Examples:
$\{nil \rightarrow nil,$
$(A) \rightarrow (A),$
$(A\ B) \rightarrow (B\ A),$
$(A\ B\ C) \rightarrow (C\ B\ A)\}$
Traces:
$f_1(x) = nil$
$f_2(x) = cons(car(x), nil)$
$f_3(x) = cons(cadr(x), cons(car(x), nil))$
$f_4(x) = cons(caddr(x), cons(cadr(x), cons(car(x), nil)))$
Differences:
$f_2(x) = cons(car(x), f_1(x))$
$f_3(x) = cons(cadr(x), f_2(x))$
$f_4(x) = cons(caddr(x), f_3(x))$
Variable Introduction:
$g_1(x, y) = y$
$g_2(x, y) = cons(car(x), y)$
$g_3(x, y) = cons(cadr(x), cons(car(x), y))$
$g_4(x, y) = cons(caddr(x), cons(cadr(x), cons(car(x), y)))$
Pairs of Differences: ($\alpha$ = anything)
$g_2(x, y) = \{cons(car(x), g_1(x, y)), g_1(\alpha, cons(car(x), y))\}$
$g_3(x, y) = \{cons(cadr(x), g_2(x, y)), g_2(cdr(x), cons(car(x), y))\}$
$g_4(x, y) = \{cons(caddr(x), g_3(x, y)), g_3(cdr(x), cons(car(x), y))\}$
Recurrence Relation:
$f_i(x) = g_i(x, nil), i \geq 1$
$g_1(x, y) = y$
$g_{k+1}(x, y) = g_k(cdr(x), cons(car(x), y))$
Recursive Program:

$$\begin{array}{ll} reverse(x) \leftarrow & G(x, nil) \\ G(x, y) \leftarrow & (atom(x) \rightarrow y, \\ & T \rightarrow G(cdr(x), cons(car(x), y))) \end{array}$$

*Figure 6.14.* Traces for the *reverse* Problem

**Definition 6.35 (Extension of Summers' Program Scheme)**

$$\begin{array}{ll} F(x) \leftarrow f(x, i(x)) & \\ f(x, z) \leftarrow & (p_1(x) \rightarrow f_1(x, z), \\ & \cdots, \\ & p_k(x) \rightarrow f_k(x, z), \\ & T \rightarrow h(x, f(b(x), g(x, z)))) \end{array}$$

*where:*
*$b$ is a basic function,*
*$i(x)$ is an initialization function,*
*$h, g$ are programs satisfying the scheme.*

The greater complexity of to be synthesized programs is mainly obtained by the matching algorithm introduced by this group, called BMWk (for "Boyer, Moore, Wegbreit, Kodratoff") which allows to determine how many variables

must be introduced in the to be constructed function to obtain regular differences between traces. An overview of this work is given in Smith (1984).

Another extension was proposed by Wysotzki (1983). He reformulated the second step of program synthesis, that is, recurrence detection in finite programs, on a more abstract level: The to be synthesized programs are characterized by term algebras instead of Lisp functions. Our work is based on this extension and presented in detail in chapter 7. An additional contribution of Wysotzki was to demonstrate that induction of recursive programs can be applied to planning problems (such as the *clearblock* problem presented in sect. 1.4.1 in chap. 3). This new area of application becomes possible because, when representing programs over term algebras, the choice of the interpreter function is free. Therefore, function symbols might be interpreted by operations defined in a planning domain (such as *put* or *puttable*, see chapter 2). Our work on learning recursive control rules for planning (see chapter 8) is based on Wysotzki's ideas. Later on Wysotzki (1987) presented an approach to realize the first step of program synthesis, that is, constructing finite programs, using universal planning. Extensions of this ideas are also presented in chapter 8.

### 3.4.3 BIERMANN'S FUNCTION MERGING APPROACH

An approach which realizes the second step of synthesis differently from Summers' approach was proposed by Biermann (1978). While Summers-like synthesis is based on detecting regularities in differences between *pairs* of traces, Biermann works with a single trace. A trace is represented by a set of non-recursive function calls and regularities are searched in differences between these functions. The basic Lisp operations used for generating traces from input/output examples are the same as in Summers' approach. The program scheme for the to be synthesized functions, called semi-regular Lisp scheme, is:

**Definition 6.36 (Semi-Regular Lisp-Scheme)**

$$\begin{aligned} F_i(x) \leftarrow \quad & (p_1(x) \rightarrow f_1(x), \\ & \cdots, \\ & p_k(x) \rightarrow f_k(x), \\ & T \rightarrow f_{k+1}(x)) \end{aligned}$$

*where:*
$p_1, \ldots p_k$ *are predicates of the form* $atom(b_i(x))$,
$b_i$ *is a basic function,*
$f_1, \ldots, f_k$ *have one of the following forms:* $nil$, $x$, $F_j(car(x))$, $F_j(cdr(x))$, $cons(F_g(x), F_h(x))$,
*and it holds:* $p_j(x) = atom(b_{j+1}(x)) \Rightarrow b_{j+1} = b_j \circ w$ *with* $w$ *as basic function.*

*Table 6.13.* Constructing a Regular Lisp Program by Function Merging

Example: $((a.b).(c.d)) \rightarrow ((d.c).(b.a))$
Resulting Semi-Trace: $cons(cons(cddr(x), cadr(x)), cons(cdar(x), caar(x)))$
Trace:
$f_1(x) = cons(f_2(x), f_3(x))$
$f_2(x) = f_4(cdr(x))$
$f_3(x) = f_5(car(x))$
$f_4(x) = cons(f_6(x), f_7(x))$
$f_5(x) = cons(f_8(x), f_9(x))$
$f_6(x) = f_{10}(cdr(x))$
$f_7(x) = f_{11}(car(x))$
$f_8(x) = f_{12}(cdr(x))$
$f_9(x) = f_{13}(car(x))$
$f_{10}(x) = f_{11}(x) = f_{12}(x) = f_{13}(x) = x$
Resulting Program:

$$\begin{array}{lll} g_1(x) \leftarrow & (atom(x) \rightarrow x, \\ & \quad T \rightarrow cons(g_2(x), g_3(x))) \\ g_2(x) \leftarrow & g_1(cdr(x)) \\ g_3(x) \leftarrow & g_1(car(x)) \end{array}$$

A set of functions corresponding to such schemes, is a program, called *regular Lisp program*.

In the first step, a semi-trace is calculated from the given example, using the algorithm $ST$ (see tab. 6.11). This semi-trace is then transformed in a trace by assigning a function definition to each sub-expression. An example is given in table 6.13 (see also Smith, 1984; Flener, 1995). Note, that the given input/output example represents nested *cons*-pairs. A *cons*-pair has the form *form((a.b))* = $(\omega\ \omega)$.

A regular Lisp program can be represented as directed graph where each node represents a function. A directed arc represents a part of a conditional expression contained in the start node. Such a conditional expression is a pair of predicate and *cons*-expression. The goal nodes of an arc are function calls appearing within the *cons*-expression. An example is given in the last line (*e*) in figure 6.15.

The construction of the graph which represents the searched for program is realized by search with backtracking. The given trace is processed top-down for the given function calls $f_i$ and for each function call it is checked whether it can be merged (is identical) with an already existing node in the graph. Merging results in cycles in the graph, representing recursion. If no merge is possible, a new arc is introduced, representing a new conditional expression.

We demonstrate this method for the trace given in table 6.13 in figure 6.15: Initially, a node $g_1$ is created, representing function $f_1$. Function $f_1$ calls two, possibly different functions, that is, two arcs, representing two conditional expressions, are introduced. Function call $f_2$ could be either identical with $g_1$ (case *b.1*) or not (case *b.2*). The first hypothesis fails and case *b.2* is selected.

Function call $f_3$ could be either identical to $g_1$ (fails) or to $g_2$ (fails) or represent a new case. Because it cannot be merged with $f_1$ or $f_2$, a new node is introduced (case *c.3*). Function calls $f_4$ and $f_5$ can be merged with $f_1$ and therefore can be represented by node $g_1$ (case *d*). As a consequence, $f_6$ and $f_8$ are identified with $g_2$ and $f_7$ and $f_9$ with $g_3$. Identification for $f_{10}$ to $f_{13}$ fails and a new arc is introduced. The resulting graph represents the final recursive program given in table 6.13.

In contrast to Summers' approach, function merging is not restricted to linear recursion. A drawback is, that Biermann's method corresponds to an identification by enumeration approach (see sect. 3.1.2) and therefore is not very efficient.

### 3.4.4 GENERALIZATION TO N AND PROGRAMMING BY DEMONSTRATION

A subfield of inductive program synthesis is to only consider how traces or protocols can be generalized to recursive programs (Bauer, 1975). Such approaches only address the second step of synthesis – detecting recurrence relation in traces. This area of research is called *programming by demonstration* (Cypher, 1993; Schrödl and Edelkamp, 1999) or *generalization-to-n* (Shavlik, 1990; Cohen, 1992). As described for Summers' recurrence detection method above, these approaches are based on the idea that traces can be considered as the $k$-th unfolding of an unknown recursive program. That is, in contrast to the folding technique in program transformation (see sect. 2.2.1), in inductive synthesis folding cannot rely on an already given recursive function!

**Programming by Demonstration with *Tinker*.** Programming by demonstration is for example used for tutoring of beginners in Lisp programming. The system *Tinker* (Lieberman, 1993) can create Lisp programs from user interactions with a graphical interface. For example, the user can manipulate blocks-worlds with operations *put* and *puttable* (see chapter 2) and the system creates Lisp code realizing the operation sequences performed by the user. In the most simple case, generalization is performed by replacing constants by variables. *Tinker* can induce simple linear recursive functions, such as the *clearblock* function (see sect. 1.4.1 in chap. 3) from multiple examples (see fig. 6.16). Conditional expressions are constructed by asking the user about differences between examples.

**Generalizing Traces using Grammar Inference.** A recent approach to programming by demonstration was presented by Schrödl and Edelkamp (1999). The authors describe how the grammar inference algorithm ID (Angluin, 1981) can be applied to induce control structures from traces. As an example they demonstrate how the loops for *bubble-sort* can be learned from user traces

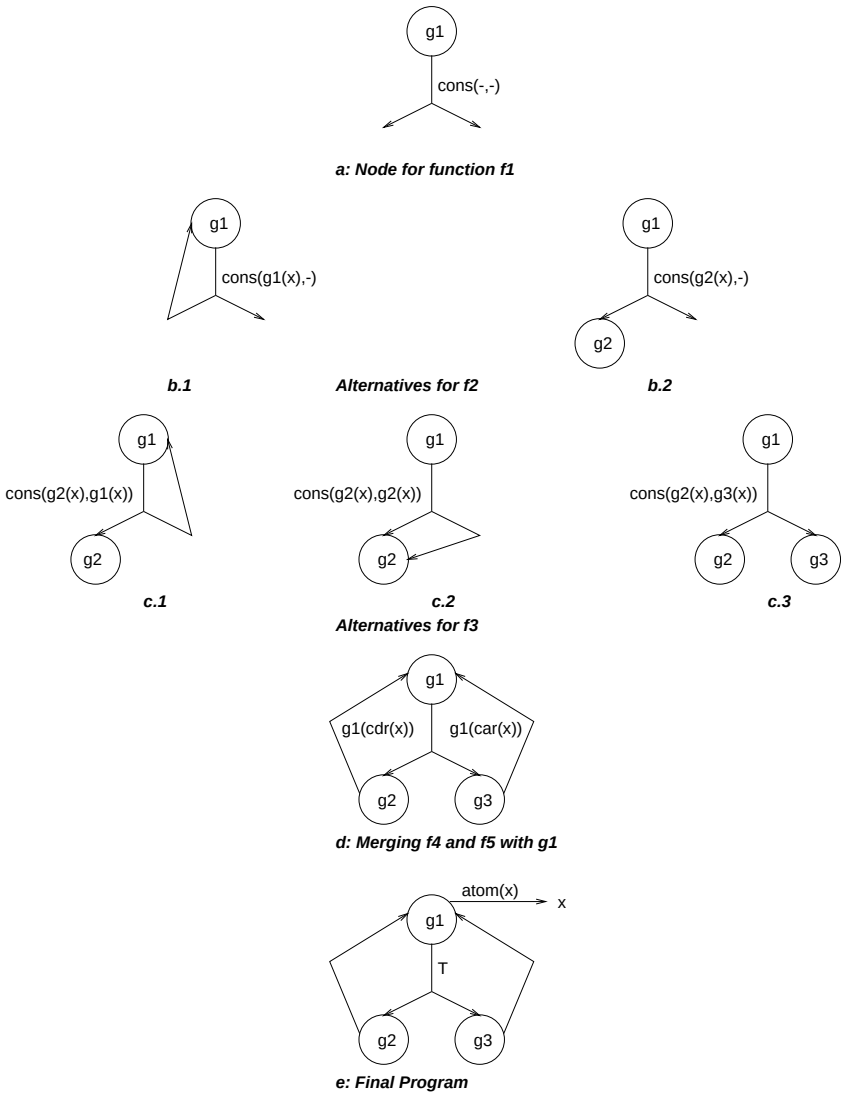



*Figure 6.15.* Synthesis of a Regular Lisp Program

for sorting lists with a small number of elements using the *swap*-operation. The trace is enriched by built-in knowledge about conditional branches. A deterministic finite automaton (DFA) is learned from the example traces and membership queries to the user. The automaton accepts all prefixes of traces (that is, truncations of the original traces) of the target program.

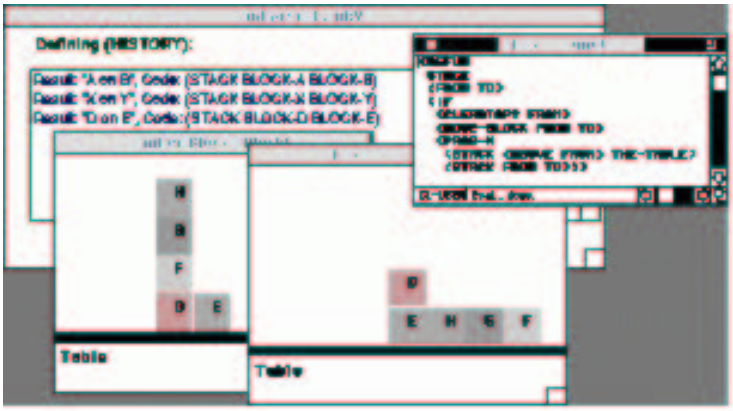



*Figure 6.16.* Recursion Formation with *Tinker*

**Explanation-Based Generalization-to-n.** An approach related to program synthesis from examples is explanation based learning or generalization (EBG) (Mitchell et al., 1986). A (possibly recursive) concept description is learned from a small set of examples based on (a large amount) of background knowledge. The background knowledge (domain theory) is used to generate an "explanation" why a given example belongs to the searched for concept. EBG is characterized as *analytical learning*: The constructed hypotheses do not extend the deductive closure of the domain theory. Typically, *Prolog* is used as hypothesis language (Mitchell, 1997, chap. 11). EBG-techniques are for example used in the planning system *Prodigy* for learning search control rules (see sect. 5.2 in chap. 2).

An extension of EBG to generalization-to-n was presented by Shavlik (1990). In a first step, explanations are generated and in a second step it is tried to detect repeatedly occuring sub-structures in an explanation. If such sub-structures are found, the explanation is transformed into a recursive rule.

# 4 FINAL COMMENTS

## 4.1 INDUCTIVE VERSUS DEDUCTIVE SYNTHESIS

As we have seen in this section, both deductive and inductive approaches heavily depend on search. In the theorem proving approaches, it is searched for such a sequence of application of programming axioms that an axiom defining the searched for input/output relation becomes true. In the program transformation approaches, it is searched for a sequence of transformation rules which, applied to a given input specification, results in an efficiently executable program. In genetic programming, it is searched for a syntactically correct

program which works correctly for all presented examples. In inductive logic programming, it is searched for a logical program which covers all positive and none of the negative examples of the to be induced relation. In inductive functional program synthesis, it is searched for regularities in traces which allow to interpret the traces as $k$-th approximation of a recursive function.

For all approaches holds, that in general search is inefficient, and therefore must be restricted by knowledge. One possibility to restrict search in deductive as well as inductive approaches is to provide program schemes: For example, the program transformation system KIDS, relies on sophisticated program schemes for *divide-and-conquer*, *local* and *global search*. In inductive logic programming, a syntactic bias is introduced by restricting the class of clauses which can be induced. In inductive functional program synthesis, the class of to be inferred functions is characterized by schemes.

We have seen for transformational synthesis that the step of transforming a non-executable specification into an (inefficient) executable program cannot be performed automatical. Instead, this step depends heavily on interaction with a user. A similar bottleneck is the first step in inductive functional program synthesis – transforming input/output examples into traces. In the systems described in this section, this problem was reduced by allowing only lists (and not numbers) as input and by considering only the structure but not the content of the input examples. If these constraints are relaxed, there is no longer a unique trace for characterizing the transformation of an input example in the desired output but potentially infinitely many and it depends on the generated traces whether a recurrence relation can be found or not. In contrast, the second step of synthesis – identifying regularities in differences between traces – can be performed straight-forward by pattern-matching. The same is true for program transformation – there are several approaches to automatic optimization of an executable program.

## 4.2 INDUCTIVE FUNCTIONAL VERSUS LOGIC PROGRAMMING

There exists no systematic comparison of inductive logic and inductive functional programming in literature. A presupposition would be to provide a formal characterization of the class of programs which can be inferred within each approach. In inductive functional programming, there exist some preliminary proposals for such a characterization (Le Blanc, 1994). Possibly, the theoretical framework of grammar inference could be used to describe what class of recursive programs can be induced on the basis of what information. Currently, we can offer only some informal observations to compare inductive logic and functional programming:

ILP typically depends on positive and negative examples, where negative examples are used to remove literals from the to be learned clauses. Inductive

functional programming uses positive examples only, but these examples must represent the first $k$ elements of a totally ordered input domain. Because of this restriction, in inductive functional programming much less examples are needed – ranging from one to about four – than in ILP, where typically more than ten positive examples must be presented.

In inductive functional programming, the resulting recursive function is independent of the sequence in which the input/output examples are presented, while in ILP different programs can result for different example sequences. The reason is, that inductive functional programming bases the construction of predicates which discriminate between examples on knowledge about the order of the input domain, while ILP stepwise includes positive examples to construct a hypothesis.

Inductive functional programming works in a two step process – first transforming input/output examples into traces and then identifying regularities in traces. This corresponds to a bottom-up, data-driven approach to learning. ILP, regardless whether it works with top-down refinement or bottom-up generalization, incrementally modifies an initial hypothesis by adding or deleting literals.

In inductive functional programming, the resulting program is executable while this must not be true for ILP: typically neither a base case for a recursive clause is induced, nor is a order for the literals in the body provided. It is a fundamental difference between logic and functional programs that for logic programs control flow is provided by the interpreter strategy (e. g., SLD-resolution), while functional programs implement a fixed control structure. Furthermore, functional programs typically have one parameter less than logic programs. In the first case, a function transforms a given input into an output value. In the second case, an input/output relation is proved and it is possible to either construct and output such that the relation holds for the given input or to construct an input such that the relation holds for the desired output.

Because functions are a subset of relations, namely relations which are defined for all inputs and have a unique output, it can be conjectured that in general, induction of functions is a harder problem than induction of relations.

Chapter 7

# FOLDING OF FINITE PROGRAM TERMS

*"It fits into the pattern, I think." "Ah," said Mr. Rattisbon who knew Alleyn. "The pattern. Your pet theory, Chief Inspector." "Yes, Sir, my pet theory. I hope you may provide me with another lozenge in the pattern."*

—Ngaio Marsh, *Death of a Peer*, 1940

In this chapter we present our approach to folding finite program terms in recursive programs. That is, we address inductive program synthesis from traces. This problem is researched in inductive synthesis of functional programs – as second step of synthesis – and in programming by demonstration (see chap. 6). Traces can be provided to the system by the system user, they can be recorded from interactions of a user with a program, or they can be constructed over a set of input/output examples. In the next chapter (chap. 8) we will describe how finite programs can be generated by planning. Folding of finite programs into recursive programs is a complex problem itself. Our approach allows to fold recursive programs which can be characterized by sets of context-free term rewrite rules. It allows to infer programs consisting of a set of recursive equations, and to deal with interdependent and hidden parameters. Traces can be generic or over instantiated parameters and they can contain incomplete paths. That is, at least for the second step of program synthesis, our approach is more powerful than other approaches discussed in literature (see sect. 3 in chap. 6).

In the following we will first introduce terms and recursive program schemes as background for our approach (sect. 1), then we will formulate the synthesis

problem (sect. 2), afterwards we present theoretical results and algorithms (sect. 3), and finally, we give some examples (sect. 4).[1]

In appendix A4 we give a short overview of folding algorithms we realized over the last years and in appendix A5 we give some details about the current implementation which is based on the approach presented in this chapter.

# 1 TERMINOLOGY AND BASIC CONCEPTS

In contrast to other approaches to the synthesis of recursive programs, the approach presented here is independent of a given program language. Instead, programs are characterized as elements of some arbitrary term algebra as proposed by Wysotzki (1983), based on the theory of recursive programs developed by Courcelle and Nivat (1978). A similar proposal was made by Le Blanc (1994) who formalized inductive program synthesis in a term rewriting framework. The goal of folding is to induce a recursive program scheme from some arbitrary finite program term. Our approach is purely syntactically, that is, it works on the structure of given program terms, independent of the interpretation of the symbols over which a term is constructed. The overall idea is to detect a recurrent relation in a finite program term. Our approach is an extension of Summers' synthesis theorem which was presented in detail in section 3.4.1 in chapter 6.

We first introduce some basic concepts and notations for terms, mostly following (Dershowitz and Jouanaud, 1990), then we introduce first order patterns and anti-unification (Burghardt and Heinz, 1996), and finally, recursive program schemes (Courcelle and Nivat, 1978).

## 1.1 TERMS AND TERM REWRITING

**Definition 7.1 (Signature)** *A signature $\Sigma$ is a set of (function) symbols with $\alpha : \Sigma \to \mathcal{N}$ giving the arity of a symbol. The set $\Sigma^n \subseteq \Sigma$ represents all symbols with fixed arity $n$, where symbols in $\Sigma_0$ represent constants. A signature can be defined over a set of variables $X$ with $X \cap \Sigma = \emptyset$.*

Although we are only concerned with the syntactical structure of terms built over a signature, in table 7.1 we give a sample of function symbols and their usual interpretation which we will use in illustrative examples. For better readability, we will sometimes represent numbers or lists in their usual notation, instead as constructor terms (e. g., 3 instead of *succ(succ(succ(0)))* or *[9, 4, 7]* instead of *cons(9, cons(4, cons(7, nil)))*).

[1] This chapter is based on the previous publications Schmid and Wysotzki (1998) and Mühlpfordt and Schmid (1998), and the diploma theses of Mühlpfordt (2000) and Pisch (2000). Formalization, proof, and implementation are mainly based on the thesis of Mühlpfordt (2000).

*Table 7.1.* A Sample of Function Symbols

| | |
|---|---|
| $0^0$ | natural number zero |
| $1^0$ | natural number one |
| $nil^0$ | empty list, truth value "false" |
| $T^0$ | truth value "true" |
| $succ^1$ | successor of a natural number |
| $pred^1$ | predecessor of a natural number |
| $head^1$ | head of a list |
| $tail^1$ | tail of a list |
| $eq0^1$ | test whether a natural number is equal zero |
| $eq1^1$ | test whether a natural number is equal one |
| $empty^1$ | test whether a list is empty |
| $cons^2$ | insertion of an element in front of a list |
| $+^2$ | addition of two numbers |
| $*^2$ | multiplication of two numbers |
| $eq^2$ | equality-test for two symbols |
| $<^2$ | test whether a natural number is smaller than another |
| $if^3$ | conditional "if $x$ then $y$ else $z$" |

**Definition 7.2 (Term)** *For the set $\mathcal{T}_\Sigma(X)$ of terms over a signature $\Sigma$ and a set of variables $X$ holds:*

1. $X \subseteq \mathcal{T}_\Sigma(X)$,
2. $\Sigma_0 \subseteq \mathcal{T}_\Sigma(X)$, *and*
3. $t_1, \ldots, t_n \in \mathcal{T}_\Sigma(X), f \in \Sigma_n \Rightarrow f(t_1, \ldots, t_n) \in \mathcal{T}_\Sigma(X)$.

*Terms without variables are called* ground terms *and the set of all ground terms is denoted as $\mathcal{T}_\Sigma$. For a term $t = f(t_1, \ldots, t_n)$ the terms $t_i (i = 1 \ldots n)$ are called* sub-terms *of $t$. Mapping $\mathbf{var} : \mathcal{T}_\Sigma(X) \rightarrow X$ returns all variables in a term.*

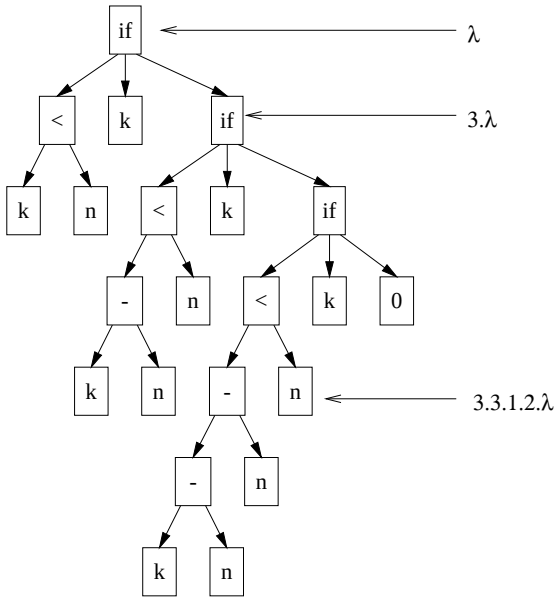
In the following we use the expressions term, program term, and tree as synonyms. Each symbol contained in a term is called a node in the term.

**Definition 7.3 (Position)** *Positions in a term $t \in \mathcal{T}_\Sigma(X)$ are defined by:*

- *$\lambda$ is the root-position of $t$,*
- *if $t = f(t_1, \ldots, t_n)$ and $u$ is a position in $t_i$ then $i.u$ is a position in $t$.*

An example for a term together with an illustration of positions in the term is given in figure 7.1.

**Definition 7.4 (Composition and Order of Positions)** *The composition $v \circ w$ of two positions $v$ and $w$ is defined as $u.w$ if $v = u.\lambda$.*




*t = if(<(k,n),k,if(<(-(k,n),n),k,if(<(-(-(k,n),n),n),k,0)))*

*Figure 7.1.* Example Term with Exemplaric Positions of Sub-terms

*A partial order over positions is defined in the following way:*
*The relation $v \leq w$ is true iff*

- $v = w$, *or*
- *exists a position* $u$ *with* $v \circ u = w$.

**Definition 7.5 (Sub-term at a Position)** *A sub-term of a term* $t \in \mathcal{T}_\Sigma(X)$ *at a position* $u$ *in* $t$ *(written* $t|_u$*) is defined as*

- $t|_\lambda = t$,
- *if* $t = f(t_1, \ldots, t_n)$ *and* $u$ *a position in* $t_i$, *then* $t|_{i.u} = t_i|_u$.

**Definition 7.6 (Positions of a Node)** *Function* $\mathbf{node} : \mathcal{T}_\Sigma(X) \times$ Position $\rightarrow$ $\Sigma$ returns the symbol at a fixed position in a term:
$\mathbf{node}(t, u) = f$ with $t \in \mathcal{T}_\Sigma(X)$, a position $u$ with $t|u = f(t_1, \ldots, t_n)$, and $f \in \Sigma$.
The set of all positions at which a fixed symbol $f$ appears in a term is denoted $\mathbf{pos}(t, f)$ and it holds $\forall w \in \mathbf{pos}(t, f) : \mathbf{node}(t, w) = f$.
The definitions can be extended to $\Sigma \cup X$. If necessary, the definition of $\mathbf{node}$ can be extended to return additionally the arity of a symbol. With $\mathbf{pos}(t)$ we

refer to the set of all positions for a term $t$ and with $\mathbf{lpos}(t)$ we refer to the set of all positions of leaf-nodes in a term.

For the term in figure 7.1, a sub-term is for example $t|_3.3.1.\lambda = <(-(-(k, n), n), n)$ and a set of positions is for example $\mathbf{pos}(t, <) = \{1.\lambda, 3.1.\lambda, 3.3.1.\lambda\}$.

**Definition 7.7 (Replacement of a Term at a Position)** *The replacement of a sub-term $t|_w$ by a term $s \in \mathcal{T}_\Sigma(X)$ in a term $t \in \mathcal{T}_\Sigma(X)$ is written as $t[w \leftarrow s]$.*

**Definition 7.8 (Substitution)** *A substitution $\sigma : X \to \mathcal{T}_\Sigma(X)$ is a mapping of variables to terms. The extension of substitution for terms is $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. Substitutions over a term are enumerated as $t\{x_1, \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ or $t[t_1, \ldots t_n]$ for short.*

**Definition 7.9 (Term Rewrite System)** *A term rewrite system over a signature $\Sigma$ is a set of pairs of terms $\mathcal{R} \subseteq \mathcal{T}_\Sigma(X) \times \mathcal{T}_\Sigma(X)$. The elements $(l, r)$ of $\mathcal{R}$ are called rewrite rules and are written as $l \to r$.*

**Definition 7.10 (Term Rewriting)** *Let $\mathcal{R}$ be a term rewrite system and $t, t' \in \mathcal{T}_\Sigma(X)$ two terms. Term $t'$ can be derived in one rewrite step from $t$ using $\mathcal{R}$ ($t \to_\mathcal{R} t'$), if there exists a position $u$ in $t$, a rule $l \to r \in \mathcal{R}$, and a substitution $\sigma : X \to \mathcal{T}_\Sigma(X)$, such that holds*

- $t|_u = \sigma(l)$,
- $t' = t[u \leftarrow \sigma(r)]$.

*$\mathcal{R}$ implies a rewrite relation $\to_\mathcal{R} \subseteq \mathcal{T}_\Sigma(X) \times \mathcal{T}_\Sigma(X)$ with $(t, t') \in \to_\mathcal{R}$ if $t \to_\mathcal{R} t'$. The reflexive and transitive closure of $\to_\mathcal{R}$ is $\overset{*}{\to}_\mathcal{R}$.*

## 1.2 PATTERNS AND ANTI-UNIFICATION

A term $t = \sigma(p)$ which can be generated by substitutions over a term $p$ is a specialization of $p$ and $p$ is called (first order) pattern of $t$.

**Definition 7.11 (First Order Pattern)** *A term $p \in \mathcal{T}_\Sigma(Y)$ is called first order pattern of a term $t \in \mathcal{T}_\Sigma(X)$, if there exists a subsumption relation $p \leq t$ with $\sigma : Y \to \mathcal{T}_\Sigma(X)$ and $\sigma(p) = t$. The sets of variables $Y$ and $X$ are disjoint.*
*A pattern $p$ of a term $t$ is called* trivial, *if $p$ is a variable and* non-trivial *otherwise.*

**Definition 7.12 (Order over Terms)** *For two terms over the same signature $p, t \in \mathcal{T}_\Sigma(X)$ holds $p \leq t$ if $p$ is a pattern of $t$ and $p < t$ if there exists no variable renaming such that $p$ and $t$ are unifiable.*

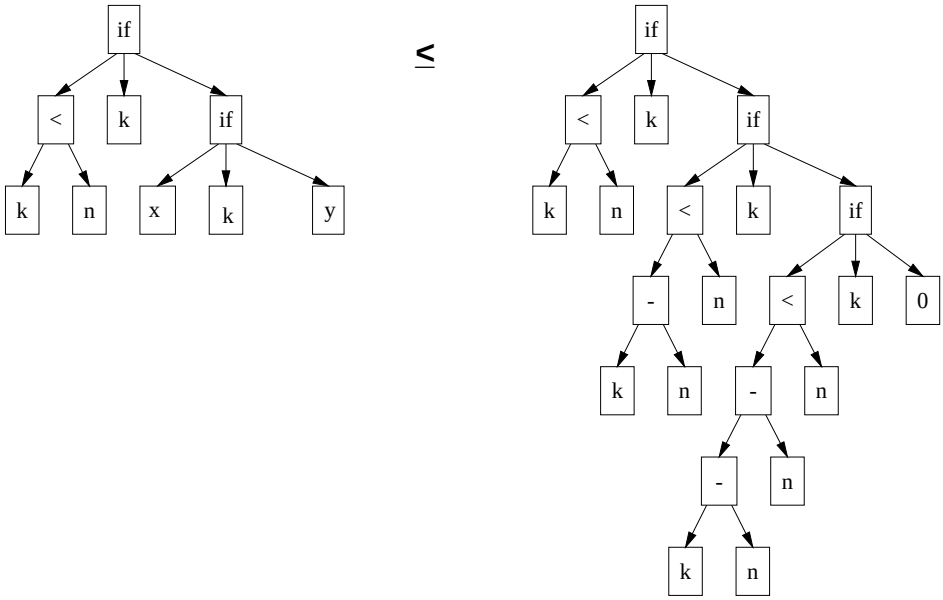



*Figure 7.2.* Example First Order Pattern

**Definition 7.13 (Maximal Pattern)** *A term $p$ is called maximal (first order) pattern of a term $t$, if $p \leq t$ and there exists no term $p'$ with $p < p'$ and $p' \leq t$.*

An example for a first order pattern of a term is given in figure 7.2. For all non-trivial patterns holds that terms which can be subsumed by a pattern share a common prefix.

A (maximal) pattern of two terms can be constructed by syntactic anti-unification of these terms.

**Definition 7.14 (Anti-Unificator)** *A term $p$ is called (first-order) anti-unificator of two terms $t$ and $t'$, if $p \leq t$ and $p \leq t'$ and if for all terms $p'$ with $p' \leq t$ and $p' \leq t'$ is $p' \leq p$. That is, the anti-unificator is the most specific generalization of two terms and it is unique except for variable renaming.*

**Definition 7.15 (Anti-Unification)** *Anti-unification $\sqcap : \mathcal{T}_\Sigma(X) \times \mathcal{T}_\Sigma(X) \rightarrow \mathcal{T}_\Sigma(X \cup Y)$ is defined as:*

- $f(t_1, \ldots, t_n) \sqcap f(t'_1, \ldots, t'_n) = f(t_1 \sqcap t'_1, \ldots, t_n \sqcap t'_n)$,
- $f(t_1, \ldots, t_n) \sqcap f'(t'_1, \ldots, t'_m) = \varphi(f(t_1, \ldots, t_n), f'(t'_1, \ldots, t'_m))$ *for* $f \neq f'$

*with $\varphi : \mathcal{T}_\Sigma(X) \times \mathcal{T}_\Sigma(X) \rightarrow Y$ as injective mapping of terms in a (new) set of variables.*

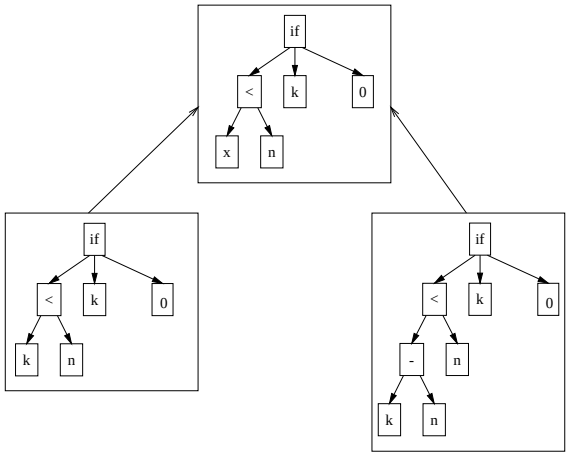



*Figure 7.3.* Anti-Unification of Two Terms

Mapping $\varphi$ guarantees that the mapping of pairs of terms to variables is unique in both directions. Identical sub-term pairs are replaced by the same variable over the complete term (Lassez, Maher, and Marriott, 1988).

Anti-unification can be extended from pairs of terms to sets of terms.

**Definition 7.16 (Anti-Unification of Sets of Terms)** *Anti-unification of a set of terms $T$ $\bigsqcap : \mathcal{T}_\Sigma(X)^+ \rightarrow \mathcal{T}_\Sigma(X \cup Y)$ is defined as:*

$$\bigsqcap T = \begin{cases} t & if\ T = \{t\} \\ (\bigsqcap(T \setminus \{t\})) \sqcap t & otherwise. \end{cases}$$

First order anti-unification is distributive, that is, terms in $T$ can be anti-unified in an arbitrary order (Lassez et al., 1988).

An example for anti-unification is given in figure 7.3.

## 1.3 RECURSIVE PROGRAM SCHEMES

### 1.3.1 BASIC DEFINITIONS

The notion of a program scheme allows us to characterize the class of recursive functions which can be induced in program synthesis.

**Definition 7.17 (Recursive Program Scheme)** *Let $\Sigma$ be a signature and $\Phi = \{G_1, \ldots, G_n\}$ a set of function variables with $\Sigma \cap \Phi = \emptyset$ and arity $\alpha(G_i) = m_i > 0$. A recursive program scheme (RPS) $\mathcal{S}$ is a pair $(\mathcal{G}, t_0)$ with $t_0 \in$*

*$\mathcal{T}_{\Sigma\cup\Phi}(X)$ and $\mathcal{G}$ as a system of $n$ equations:*

$$\mathcal{G} = \begin{array}{ll} \langle G_1(x_1,\ldots,x_{m_1}) & = t_1, \\ \vdots & \\ G_n(x_1,\ldots,x_{m_n}) & = t_n\rangle \end{array}$$

*with $t_i \in \mathcal{T}_{\Sigma\cup\Phi}(X), i = 1\ldots n$.*

An RPS corresponds to an abstract functional program with $t_0$ as initial program call ("main program") and $\mathcal{G}$ as a set of (recursive) functions ("sub-programs"). The left-hand side of an equation in $\mathcal{G}$ is called program head, the right-hand side program body. The parameters $x_i$ in a program head are pairwise different. The body $t_i$ of an equation $G_i$ can contain calls of $G_i$ or calls of other equations in $\mathcal{G}$.

In the following, RPSs are restricted in the following way: For each program body $t_i$ of an equation in $\mathcal{G}$ holds

- $\mathbf{var}(t_i) = \{x_1,\ldots,x_{m_i}\}$, that is, all variables in the program head are used in the program body, and
- $\mathbf{pos}(t_i, G_i) \neq \emptyset$, that is, each equation is recursive.

The first restriction does not limit expressiveness: Variables which never are used in the body of a program do not contribute to the computation of a result. The second restriction ensures that induction of an RPS from a finite program term is really due to generalization (learning) and that the resulting RPS does not just represent the given finite program itself.

An example for an RPS is given in table 7.2. Equation *ModList* checks for each element of a list of natural numbers whether it is a multiple of a number $n$. The equation is linear recursive because it contains a call of itself as argument of the *cons*-operator and it calls a second equation *Mod* which is tail-recursive, that is, a special case of linear recursion, corresponding to a loop. The program call $t_0$ in this example, is simply a call of equation *ModList*. In general, $t_0$ can be a more complex expression from $\mathcal{T}_\Sigma(X) \cup \Phi$. For example, we might only be interested whether the second element of the list is a multiple of $n$ and call $t_0$ = *head(tail(ModList(l,n)))*.

Interpretation of an RPS, for example by means of an eval-apply method (Field and Harrison, 1988), presupposes that all symbols in $\Sigma$ are available as pre-defined operations and that the variables in $t_0$ are instantiated. An RPS can be unfolded by replacing function calls by the corresponding function body where variables in the body are substituted in accordance to the parameters in the function call. Generation of program terms by unfolding is described in the next section.

Variable instantiation is a special case of substitution (see def. 7.8) where variables are replaced by ground terms:

*Table 7.2.* Example for an RPS

$\mathcal{S} = \langle \mathcal{G}, t_0 \rangle$

$\mathcal{G} =$ $\langle$
ModList(l, n) = if( empty(l),
nil,
cons( if(eq0(Mod(n, head(l))), T, nil),
ModList(tail(l), n))),
Mod(k, n) = if(<(k, n), k, Mod(−(k, n), n))
$\rangle$

$t_0 = ModList(l, n)$

**Definition 7.18 (Variable Instantiation)** *A mapping $\beta : X \to \mathcal{T}_\Sigma$ is called instantiation of variables $X$. Variable instantiation can be extended to terms: $\beta : \mathcal{T}_\Sigma(X) \to \mathcal{T}_\Sigma$ such that $\beta(f(t_1, \ldots, t_n)) = f(\beta(t_1), \ldots, \beta(t_n))$ for $f \in \Sigma$, arity $\alpha(f) = n$, and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma(X)$.*

Please note, that we speak of the (formal) *parameters* of a program to refer to its arguments. For example, the recursive equation *ModList(l, n)* has two parameters. The parameters can be *variables*, such as *l, n* ∈ *X*, or be instantiated with (ground) terms.

### 1.3.2 RPSS AS TERM REWRITE SYSTEMS

A recursive program scheme can be viewed as term rewrite system (Courcelle, 1990) or, alternatively as context-free tree-grammar (see next section).

An RPS as introduced in definition 7.17 can be interpreted as term rewrite system as introduced in definition 7.9:

**Definition 7.19 (RPS a Rewrite System)** *Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS and $\Omega$ a special symbol. The equations in $\mathcal{G}$ constitute rules $\mathcal{R}_\mathcal{S}$ of a term rewrite system. The system additionally contains rules $\mathcal{R}_\Omega$:*

- $\mathcal{R}_\mathcal{S} = \{G_i(x_1, \ldots, x_{m_i}) \to t_i \mid i \in \{1, \ldots, n\}\}$,
- $\mathcal{R}_\Omega = \{G_i(x_1, \ldots, x_{m_i}) \to \Omega \mid i \in \{1, \ldots, n\}\}$.

*We write $\overset{*}{\to}_{\Sigma,\Omega}$ for the reflexive and transitive closure of the rewrite relation implied by $\mathcal{R}_\mathcal{S} \cup \mathcal{R}_\Omega$.*

By means of the term rewrite system, the head of a user-defined equation (occuring in $t_0$ or a program body $t_i$) is either replaced by the associated body or by symbol $\Omega$ which represents the empty or undefined term.

**Definition 7.20 (Language of an RPS)** *Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS and $\beta : \mathbf{var}(t_0) \to \mathcal{T}_\Sigma$ an instantiation of the parameters in $t_0$. The set of all terms*

*$\mathcal{L}(\mathcal{S},\beta) = \{t \mid t \in \mathcal{T}_{\Sigma\cup\{\Omega\}}, \beta(t_0) \stackrel{*}{\rightarrow}_{\Sigma,\Omega} t\}$ is the language generated by the RPS with instantiation $\beta$. For a main program $t_0$ without variables, we can write $\mathcal{L}(\mathcal{S})$.*

The instantiation of all parameters in program call $t_0$ implies that the language of an RPS is a set of ground terms, containing neither object variables $X$ nor function variables $\Phi$. That is, all recursive calls are at some rewrite step replaced by $\Omega$s.

**Definition 7.21 (Language of a Subprogram)** *Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS. For an equation $G_i \in \mathcal{G}$ the rewrite relation $\rightarrow_{G_i,\Omega}$ is implied by the rules*

$$\mathcal{R}_{G_i,\Omega} = \{G_i(x_1, \ldots x_{m_i}) \rightarrow t_i, G_i(x_1, \ldots x_{m_i}) \rightarrow \Omega\}.$$

*Let $\beta : \{x_1, \ldots x_{m_i}\} \rightarrow \mathcal{T}_\Sigma$ be an instantiation of parameters of $G_i$, then the set of all terms $\mathcal{L}(G_i,\beta) = \{t \mid t \in \mathcal{T}_{\Sigma\cup\{\Omega\}\cup\Phi\setminus\{G_i\}}, \beta(G_i(x_1, \ldots, x_{m_i})) \stackrel{*}{\rightarrow}_{G_i,\Omega} t\}$ is the language generated by subprogram $G_i$ with instantiation $\beta$.*

The term given on the left side of figure 7.4 is an example for a term belonging to the language of the RPS given in table 7.2 as well as an example for a term belonging to the language of the subprogram *ModList*. The term represents an unfolding of the main program $t_0$ where the *ModList* subprogram is called. For $t_0$ = *ModList(l, n)* variables are instantiated as $l = [9, 4, 7]$ and $n = 8$.[2] Within this unfolding, there is an unfolding of the *Mod* subprogram which is called by *ModList*. At the same time, the term represents an unfolding of the subprogram *ModList* itself. The term given on the right side of figure 7.4 is an unfolding of *ModList* containing the call of the *Mod* subprogram. Neither of the two terms contain the name of the subprogram *ModList* itself.

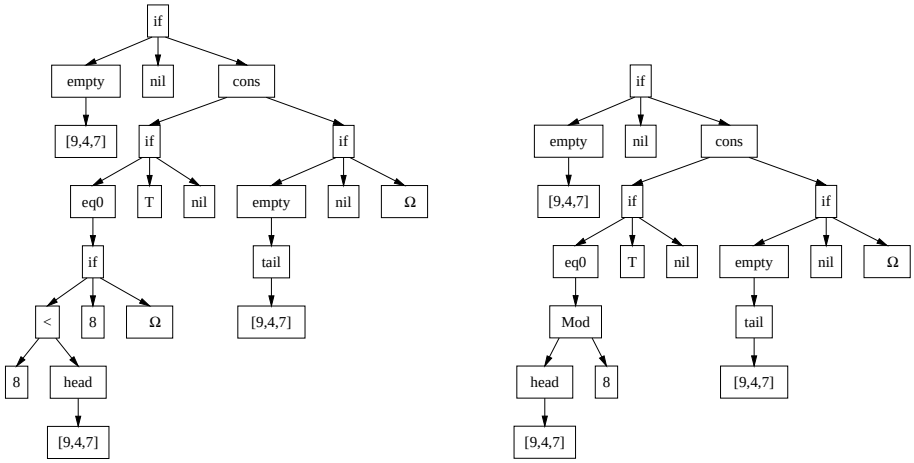
### 1.3.3 RPSS AS CONTEXT-FREE TREE GRAMMARS

In the remaining chapter, we will rely on the notion of RPSs as term rewrite systems. In this section we will shortly present an alternative, equivalent definition of RPSs as context-free tree grammars (Guessarian, 1992).

**Definition 7.22 (Context-free Tree Grammar)** *A context-free tree grammar (CFTG), $\mathcal{C} = (A, N, \Sigma, P)$, is a four-tuple of*

- *a set of terminal symbols $\Sigma$,*
- *a set of non-terminal symbols $N$ with fixed arity and $N \cap \Sigma = \emptyset$*
- *an axiom $A \in \mathcal{T}_{\Sigma\cup N}$, and*

[2] For a restricted signature with natural number 0 and successor-function only and list constructor $cons$, a natural number $i$ can be represented as $succ^i(0)$ and a list as $cons$-expression.




$$t_0 = ModList(l, n),\ \beta(l) = [9, 4, 7],\ \beta(n) = 8$$

*Figure 7.4.* Examples for Terms Belonging to the Language of an RPS and of a Subprogram of an RPS

- *a set of production rules $P$ of the form $G(x_1, \ldots, x_n) \to t$ with $G \in N$ and $\{x_1, \ldots, x_n\} \subseteq X$, where $X \cap (\Sigma \cup N) = \emptyset$ is a set of reserved variables, and $t \in \mathcal{T}_{\Sigma \cup N}(X)$.*

*If there exist for a non-terminal $G \in N$ more than one rule $G(x_1, \ldots, x_n) \to t_1, \ldots, G(x_1, \ldots, x_n) \to t_m$ we write $G(x_1, \ldots, x_n) \to t_1 \mid \ldots \mid t_m$ for short.*

The language of a context-free tree grammar consists of all words (terms) which can be derived from the axiom and which do not contain non-terminal symbols.

**Definition 7.23 (Language of a CFTG)** *Let $\mathcal{C} = (A, N, \Sigma, P)$ be an CFTG and $\overset{*}{\to}_P$ the reflexive and transitive closure of the derivation relation $\to_P$. The language $\mathcal{L}(\mathcal{C})$ of the grammar is $\mathcal{L}_{\mathcal{C}} = \{t \in \mathcal{T}_\Sigma \mid A \overset{*}{\to}_P t\}$.*

The program call $t_0$ of an RPS can be interpreted as axiom of an CFTG and the equations of an RPS as production rules. In contrast to an RPS, the axiom must contain no variables, that is, all variables in the program call must be instantiated.

**Definition 7.24 (CFTG of an RPS)** *Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS and $\beta : X \to \mathcal{T}_\Sigma$ an instantiation of variables in $t_0$ ($X = \mathbf{var}(t_0)$), then the CFTG $\mathcal{C}_{\mathcal{S},\beta}$ of*

*$\mathcal{S}$ and $\beta$ is defined as $\mathcal{C}_{\mathcal{S},\beta} = (\beta(t_0), \Phi, \Sigma, P)$ with*

$$P = \begin{array}{ll} \{G_1(x_1, \ldots, x_{m_1}) & = t_1 \mid \Omega, \\ \vdots & \\ G_n(x_1, \ldots, x_{m_n}) & = t_n \mid \Omega\}. \end{array}$$

With such a CFTG can the language generated by an RPS be defined as $\mathcal{L}(\mathcal{S}, \beta) = \mathcal{L}(\mathcal{C}_{\mathcal{S},\beta})$.

### 1.3.4 INITIAL PROGRAMS AND UNFOLDINGS

By means of the definition of an RPS as term rewrite system (see def. 7.19) and the language of an RPS induced by this system (see def. 7.20) we provide for a method to *unfold* an RPS into a term of finite length. Because all equations in an RPS must be recursive, such a finite term contains at least one symbol $\Omega$ (termination of rewriting). Now we can characterize the input in our synthesis system as a term of which we assume that it is generated by an (yet unknown) recursive program scheme.

**Definition 7.25 (Initial Program)** *An initial program is a term $t \in \mathcal{T}_{\Sigma\cup\{\Omega\}}(X)$ which contains at least one $\Omega$: $\mathbf{pos}(t, \Omega) \neq \emptyset$. The term might be defined over instantiated variables only, that is, $t \in \mathcal{T}_{\Sigma\cup\{\Omega\}}$.*

**Definition 7.26 (Order over Initial Programs)** *Let $t, t' \in \mathcal{T}_{\Sigma\cup\{\Omega\}}(X)$ be two terms. An order $t \leq_\Omega t'$ is inductively defined as*

1. *$\Omega \leq_\Omega t'$, if $\mathbf{pos}(t', \Omega) \neq \emptyset$,*
2. *$x \leq_\Omega t'$, if $x \in X$ and $\mathbf{pos}(t', \Omega) \neq \emptyset$,*
3. *$f(t_1, \ldots, t_n) \leq_\Omega f(t'_1, \ldots t'_n)$, if $\forall i \in \{1, \ldots, n\}$ holds $t_i \leq_\Omega t'_i$.*

The task of folding an initial program involves finding a segmentation and substitutions of the initial program which can be interpreted as the first $i$ elements of an unfolding of an hypothetical RPS. Therefore, we now introduce recursion points and substitution terms for an RPS.

For a given recursive equation, each occurrence of recursive call in its body constitutes a recursion point and for each recursive call the parameters in are substituted by terms:

**Definition 7.27 (Recursion Points and Substitution Terms)** *Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS, $G_i(x_1, \ldots, x_{m_i}) = t_i$ a recursive equation in $\mathcal{S}$, and $X_i = \{x_1, \ldots x_{m_i}\}$ the set of parameters of $G_i$. The non-empty set of recursion points $U_{rec}$ of $G_i$ is given by $U_{rec} = \mathbf{pos}(t_i, G_i)$ with indices (positions) $R = \{1, \ldots, |U_{rec}|\}$.*

*Table 7.3.* Recursion Points and Substitution Terms for the *Fibonacci* Function

**Recursive Equation:**

$$
\begin{aligned}
G_1(x_1) = \quad & \textit{if( eq0(}x_1\textit{),} \\
& \quad \textit{1,} \\
& \quad \textit{if( eq0(pred(}x_1\textit{)),} \\
& \quad\quad \textit{1,} \\
& \quad\quad +(G_1\textit{(pred(}x_1\textit{)),}\ G_1\textit{(pred(pred(}x_1\textit{))))))}
\end{aligned}
$$

**Recursion Points:** $U_{rec} = \{3.3.1.\lambda, 3.3.2.\lambda\}$

**Recursive Calls:** $t_1|_{3.3.1.\lambda} = G_1$*(pred(*$x_1$*))*,
$t_2|_{3.3.2.\lambda} = G_1$*(pred(pred(*$x_1$*)))*

**Substitution Terms** for $x_1$:
$\mathbf{sub}(x_1, 1) = t_1|_{3.3.1.\lambda \circ 1}$ = $G_1$*(pred(*$x_1$*))*$|_{1.\lambda}$ = *pred(*$x_1$*)*,
$\mathbf{sub}(x_1, 2) = t_1|_{3.3.2.\lambda \circ 1}$ = $G_1$*(pred(pred(*$x_1$*)))*$|_{1.\lambda}$ = *pred(pred(*$x_1$*))*

*Each recursive call of $G_i$ at position $u_r \in U_{rec}, r \in R$ in $t_i$ implies substitutions $\sigma_r : X_i \to \mathcal{T}_\Sigma(X_i)$ of the parameters in $G_i$. Let $\mathbf{sub} : X_i \times R \to \mathcal{T}_\Sigma(X_i)$ be a mapping $\mathbf{sub}(x_j, r) = \sigma_r(x_j)$ for all $x_j \in X_i$, then holds $\mathbf{sub}(x_j, r) = t_i|_{u_r \circ j}$. The terms $\mathbf{sub}(x_j, r)$ are called* substitution terms.

An example for recursion points and substitution terms for the *Fibonacci* function is given in table 7.3. *Fibonacci* is tree recursive, that is, it contains two recursive calls in its body. The positions of these calls (see def. 7.3) in the term representing the body of *Fibonacci* are $U_{rec} = \{3.3.1.\lambda, 3.3.2.\lambda\}$. For referring to these recursion points we introduce an index set $R = \{1, \ldots, |U_{rec}|\}$, that is, for *Fibonacci* is $R = \{1, 2\}$. With $u_1 \in U_{rec}$ we refer to the first recursive call and with $u_2 \in U_{rec}$ we refer to the second recursive call. In the first recursive call, the parameter $x$ is substituted by *pred(x)* and in the second recursive call, it is substituted by *pred(pred(x))*.

For a given recursive equation there is a fixed set of recursion points. This set contains a single position for linear recursion, two positions for tree recursion, and for more complex RPSs the set can contain an arbitrary number of recursion points. If a given term is generated by unfolding a recursive equation, the recursion points occur in a regular way in each unfolding. Because, theoretically, a recursive equation can be unfolded infinitely many times, there are infinitely many positions at which recursive calls occur, that is *unfolding positions*. We will describe such positions over indices in the following way:

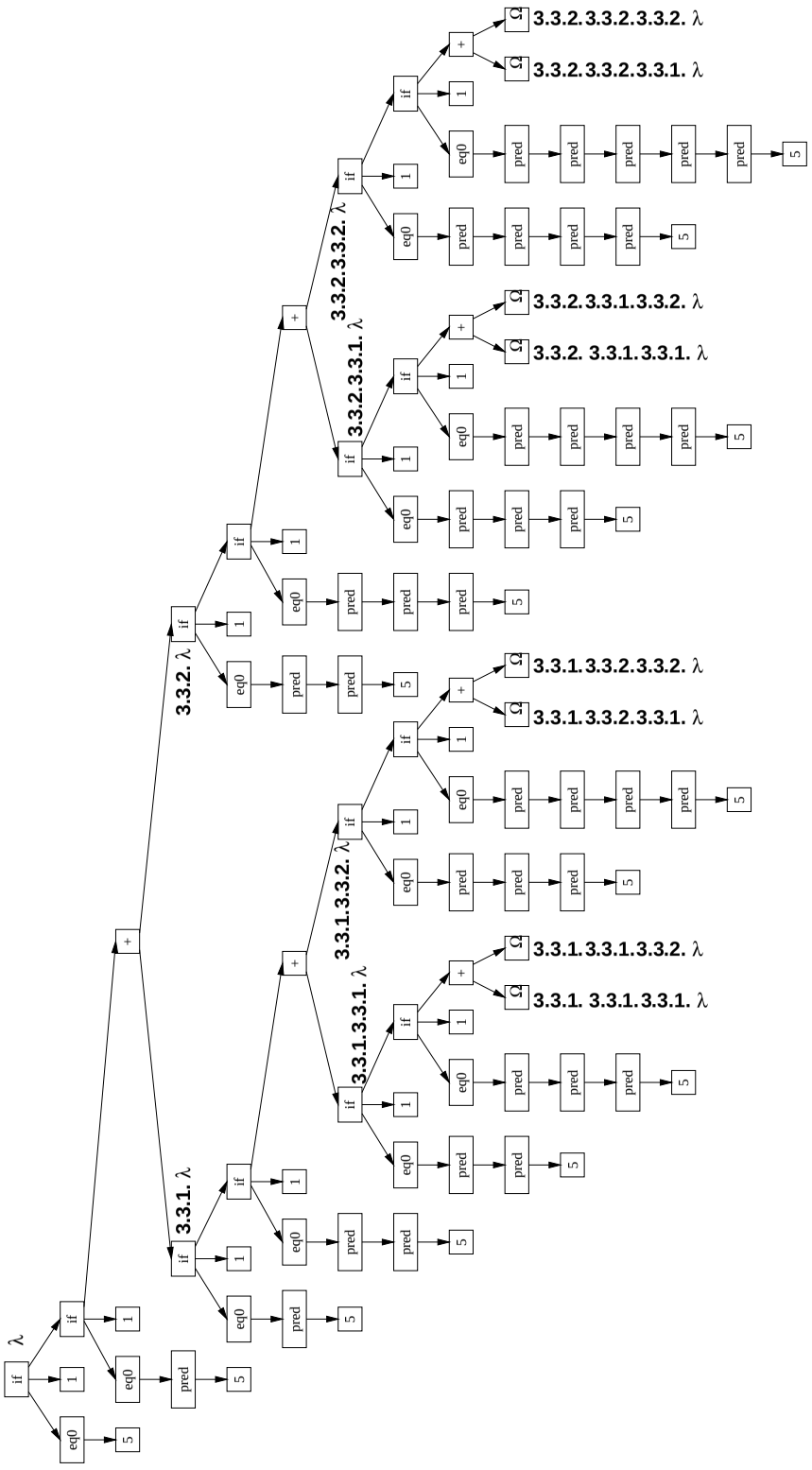



*Figure 7.5.* Unfolding Positions in the Third Unfolding of *Fibonacci*

*Table 7.4.* Unfolding Positions and Unfolding Indices for *Fibonacci*

| Unfolding | Unfolding Positions | Unfolding Indices |
|---|---|---|
| 0 | $\lambda$ | $\lambda$ |
| 1 | $3.3.1.\lambda$, $3.3.2.\lambda$ | $1.\lambda$, $2.\lambda$ |
| 2 | $3.3.1.3.3.1.\lambda$, $3.3.1.3.3.2.\lambda$, $3.3.2.3.3.1.\lambda$, $3.3.2.3.3.2.\lambda$ | $1.1.\lambda$, $1.2.\lambda$, $2.1.\lambda$, $2.2.\lambda$ |
| 3 | $3.3.1.3.3.1.3.3.1.\lambda$, $3.3.1.3.3.1.3.3.2.\lambda$, $3.3.1.3.3.2.3.3.1.\lambda$, $3.3.1.3.3.2.3.3.2.\lambda$, $3.3.2.3.3.1.3.3.1.\lambda$, $3.3.2.3.3.1.3.3.2.\lambda$, $3.3.2.3.3.2.3.3.1.\lambda$, $3.3.2.3.3.2.3.3.2.\lambda$ | $1.1.1.\lambda$, $1.1.2.\lambda$, $1.2.1.\lambda$, $1.2.2.\lambda$, $2.1.1.\lambda$, $2.1.2.\lambda$, $2.2.1.\lambda$, $2.2.2.\lambda$ |

**Definition 7.28 (Unfolding Indices)** *Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS, $G_i(x_1, \ldots, x_{m_i}) = t_i$ a recursive equation in $\mathcal{S}$, $X_i = \{x_1, \ldots x_{m_i}\}$ the set of parameters of $G_i$, and $U_{rec}$ the set of recursion points of $G_i$ with index set $R = \{1, \ldots, |U_{rec}|\}$. The set of unfolding indices $W$ constructed over $R$ is inductively defined as the smallest set for which holds:*

1. $\lambda \in W$,
2. *if $w \in W$ and $r \in R$, then $w \circ r \in W$.*

To illustrate the concept of unfolding indices, we anticipate the concept of unfolding which will be introduced in the next definition. Consider the *Fibonacci* term in figure 7.5 which represents the third syntactic unfolding of the recursive function given in table 7.3. The zero-th unfolding would be just the empty term $\Omega$ and its position is $\lambda$, that is the root of the tree. The first unfolding renders two unfolding positions, $3.3.1.\lambda$ and $3.3.2.\lambda$ which are identical with the recursion points of the *Fibonacci* function. Unfolding the function at each of these positions again by one step results in four new unfolding positions, $3.3.1.3.3.1.\lambda$, $3.3.1.3.3.2.\lambda$, $3.3.2.3.3.1.\lambda$, and $3.3.2.3.3.2.\lambda$, and so on. Just as we refer to the *recursion* points using the index set $R$, we refer to the unfolding positions using an index set $W$. The unfolding points and the corresponding indices for the first three unfoldings of *Fibonacci* are given in table 7.4. For a tree recursive structure such as *Fibonacci*, the indices grow in a treelike fashion. For example index $1.1.2.\lambda$ refers to the right unfolding position in the two left unfoldings on the levels above.

Defining the unfolding of a recursive equation is based on the positions in a term at which unfoldings are performed and on a mechanism to replace parameters in the program body:

**Definition 7.29 (Unfolding of an Recursive Equation)** *Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS, $G_i(x_1, \ldots, x_{m_i}) = t_i$ a recursive equation in $\mathcal{S}$, $X_i = \{x_1, \ldots, x_{m_i}\}$ the set of parameters of $G_i$, $U_{rec}$ the set of recursion points of $G_i$ with index set $R = \{1, \ldots, |U_{rec}|\}$, and $\mathbf{sub}(x_j, r)$ the substitution term for $x_j \in X_i$ in*

*the recursive call of $G_i$ at position $u_r \in U_{rec}, r \in R$. The set of all unfoldings $\Upsilon_i$ of equation $G_i$ over instantiations $\beta : X_i \to \mathcal{T}_\Sigma$ is inductively defined as the smallest set for which holds:*

1. *Let $\beta_\lambda = \beta$ be the instantiation of variables in unfolding $\lambda$. Then $v_\lambda = \beta_\lambda(t_i)$ and $v_\lambda \in \Upsilon_i$.*
2. *Let $v_w \in \Upsilon_i$ be an unfolding with instantiation $\beta_w$. Then for each $r \in R$ exists an unfolding $v_{w \circ r} \in \Upsilon_i$ with instantiations $\beta_{w \circ r}(x_j) = \beta_w(\mathbf{sub}(x_j, r))$ for all $x_j \in X_i$, such that $v_{w \circ r} = \beta_{w \circ r}(t_i)$.*

An unfolding is a term, which is generated in a derivation step $t \to_{\mathcal{S},\Omega} t'$ by applying a term rewrite rule $G_i(x_1, \ldots, x_{m_i}) \to t_i$ (see def. 7.19). The resulting term is an element of language $\mathcal{L}(G_i, \beta)$ (see def. 7.21). For a term $t$, which is the result of repeated application of rule $\beta(G_i(x_1, \ldots, x_{m_i}))$, the unfolding indices $w$ correspond to the sequence in which the recursive calls were replaced.

Please note, that the initial instantiation of variables $\beta_\lambda$ can be empty, that is, the variables appearing in the program body can remain uninstantiated during unfolding. For syntactical unfolding, as described above, there is no possibility to terminate unfolding for paths which could never been reached during program evaluation. For example, in figure 7.5 at each level of unfolding positions, *all* recursive calls where unfolded. It is not checked, whether one of the conditions $eq0(x_1)$ or $eq0(pred(x_1))$ is already true for the given instantiation of $x_1$. Unfolding of terms is a well introduced concept, which we already described in chapter 6 (sects. 2.2.1 and 3.4.1). In the definition above, we simply reformulated unfolding in our terminology.

The synthesis problem is performed successfully, if an RPS with a set of recursive equations can be induced which can recurrently explain a given initial program.

**Definition 7.30 ((Recursive) Explanation of an Initial Program)** *A recursive equation $G_i(x_1, \ldots, x_{m_i}) = t_i$ explains an initial program $t_{init}$, if there exist an instantiation $\beta : \{x_1, \ldots x_{m_i}\} \to \mathcal{T}_\Sigma$ of the parameters of $G_i$ and a term $t \in \mathcal{L}(G_i, \beta)$, such that $t_{init} \leq_\Omega t$. $G_i$ is a* recurrent explanation *of $t_{init}$ if furthermore exists a term $t' \in \mathcal{L}(G_i, \beta)$ which can be derived by at least two applications of $G_i(x_1, \ldots, x_{m_i}) \to t_i$ such that $t' \leq_\Omega t_{init}$.*
*An RPS $\mathcal{S} = (\mathcal{G}, t_0)$ explains an initial program $t_{init}$, if there exist an instantiation $\beta : \mathbf{var}(t_0) \to \mathcal{T}_\Sigma$ of the parameters of the program call $t_0$ and a term $t \in \mathcal{L}(\mathcal{S}, \beta)$, such that $t_{init} \leq_\Omega t$. $\mathcal{S}$ is a* recurrent explanation *of $t_{init}$ if furthermore exists a term $t' \in \mathcal{L}(\mathcal{S}, \beta)$ which can be derived by at least two applications of all rules $\mathcal{R}_\mathcal{S}$ such that $t' \leq_\Omega t_{init}$.*
*An equation/RPS explains a set of initial programs, if it explains all terms in the set and if there is a recurrent explanation for at least one term.*

Definition 7.30 states that an recursive equation or an RPS defined over a set of such explanations (subprograms) together with an initial program call explains an initial program, if some term $t$ can be derived such that the initial program is completely contained in this term (see def. 7.26). For a recurrent explanation it must hold additionally, that some term $t'$ can be derived by *repeated* unfolding (that is, at least two times) such that this term $t'$ is completely contained in the initial program.

## 2 SYNTHESIS OF RPSS FROM INITIAL PROGRAMS

Based the definition of RPSs, initial programs, unfoldings, and recurrent explanations introduced above, we are now ready to formulate the synthesis problem, which we want to solve, precisely. Before we do that, we will first discuss the relation between the fixpoint semantics of recursive functions and induction of recursive functions from initial programs and give some characteristics of RPSs.

### 2.1 FOLDING AND FIXPOINT SEMANTICS

As discussed in chapter 6, induction can be viewed as inverse process to deduction. For example, in inductive logic programming, more general clauses can be computed by *inverse* resolution, a term or clause can be generalized by *anti*-unification (Plotkin, 1969). Summers (1977) proposed that folding an initial program into a (set of) recursive equation can be seen as inverse of fixpoint semantics (Field and Harrison, 1988; Davey and Priestley, 1990). Summers synthesis theorem is given in section 3.4.1 in chapter 6.

We introduce fixpoint semantics in appendix B1. For short, fixpoint semantics allows to give a denotation to recursively defined syntactic functions. The semantics of a (continuous) function (over an ordered domain) is defined as the least supremum of a sequence of unfoldings (expansions) of this function.

For folding, we consider a given finite program term as the $i$-th unfolding $t_{init} = G^i$ of an unknown recursively defined syntactic function $G$. Finding a recursive explanation for $G_i$ as described in definition 7.30, corresponds to segmenting $G_i$ into a sequence of unfoldings $G^0, G^1, \ldots, G^i$ where $G^k = t_k[u \leftarrow G_\sigma^{k-1}]$ with $u$ as a fixed position in $t_k$ and $\sigma$ as substitution of parameters holds for $k = 1, \ldots, i$. In that case, based on the recurrence relation which holds for $G^i$, the recursive function $G = t[u \leftarrow G_\sigma]$ can be extrapolated. An example is given in table 7.5.

### 2.2 CHARACTERISTICS OF RPSS

In section 1.3.1 we already introduced two restrictions for RPSs: All variables given in the head of an recursive equation must appear in the body and

*Table 7.5.* Example of Extrapolating an RPS from an Initial Program

- $t_{init} = G^i =$
  if(<(k,n ), k, if(<(−(k, n)), k, if(<(−(−(k, n), k), k), k, Ω)))
- **Segmentation into hypothetical unfoldings of a recursive function $G$:**
  $G^0 =_{def} \Omega$
  $G^1$ = if(<(k, n),k, Ω)
  $G^2$ = if(<(k, n),k, if(<(−(k, n)),k, Ω))
  $G^3 = G^i$
- **Recurrence Relation and Extrapolation:**
  $G^1$ = if(<(k, n),k, $G^0_{[k \leftarrow -(k,n)]}$)
  $G^2$ = if(<(k, n),k, $G^1_{[k \leftarrow -(k,n)]}$)
  $G^3$ = if(<(k, n),k, $G^3_{[k \leftarrow -(k,n)]}$)
  *extrapolate:*
  $G$ = if(<(k, n),k, $G_{[k \leftarrow -(k,n)]}$)

each equation contains at least a call of itself. In the following, further characteristics of the RPSs which can be folded using our approach are given.

**Definition 7.31 (Dependency Relation of Subprograms)** *Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS with a set of function variables (names of subprograms) $\Phi$. Let $H \notin \Phi$ be a function variable (for the unnamed main program). A relation $\mathbf{calls}_\mathcal{S} \subseteq \{H\} \cup \Phi \times \Phi$ between subprograms and main program of $\mathcal{S}$ is defined as:*

$$\begin{aligned} \mathbf{calls}_\mathcal{S} \quad = \quad & \{(H, G_i) \mid G_i \in \Phi, \mathbf{pos}(t_0, G_i) \neq \emptyset\} \cup \\ & \{(G_i, G_j) \mid G_i, G_j \in \Phi, \mathbf{pos}(t_i, G_j) \neq \emptyset\}. \end{aligned}$$

*The transitive closure $\mathbf{calls}^*_\mathcal{S}$ of $\mathbf{calls}_\mathcal{S}$ is the smallest set $\mathbf{calls}^*_\mathcal{S} \subseteq \{H\} \cup \Phi \times \Phi$ for which holds:*

1. $\mathbf{calls}_\mathcal{S} \subseteq \mathbf{calls}^*_\mathcal{S}$,
2. *for all $P \in \{H\} \cup \Phi$ and $G_i, G_j \in \Phi$: If $P$ $\mathbf{calls}^*_\mathcal{S}$ $G_i$ and $G_i$ $\mathbf{calls}^*_\mathcal{S}$ $G_j$ then $P$ $\mathbf{calls}^*_\mathcal{S}$ $G_j$.*

For introducing a notion of minimality of RPSs two further restrictions are necessary:

**Only Primitive Recursion:** All recursive equations are primitive recursive, that is, there are no recursive calls in substitutions.

**No Mutual Recursion:** There are no recursive equations $G_i, G_j$ with $i \neq j$ with $G_i$ $\mathbf{calls}^*_\mathcal{S}$ $G_j$ and $G_j$ $\mathbf{calls}^*_\mathcal{S}$ $G_i$.

The first restriction was already given implicitly in definition 7.27 where $\mathbf{sub}$ maps into $\mathcal{T}_\Sigma(X_i)$. A consequence of this restriction is that we cannot infer general $\mu$-recursive functions, such as the Ackermann function. The second restriction is only syntactical since each pair of mutually recursive functions can be transformed into semantically equivalent functions which are not mutually recursive.

**Definition 7.32 (Minimality of an RPS)** *Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS. $\mathcal{S}$ is minimal if for all subprograms $G_i(x_1, \ldots, x_{m_i}) = t_i, G_i \in \Phi, X_i = \{x_1, \ldots, x_{m_i}\}$ holds:*

**No unused subprograms:** $H \ \mathbf{calls}^*_{\mathcal{S}} \ G_i$.

**No unused parameters:** *For each instantiation $\beta : X_i \to \mathcal{T}_\Sigma$ and instantiations $\beta_j : X_i \to \mathcal{T}_\Sigma, j = 1, \ldots, m_i$ constructed as*

$$\beta_j(x_k) = \begin{cases} t & k = j, t \in \mathcal{T}_\Sigma, t \neq \beta(x_j) \\ \beta(x_k) & k \neq j \end{cases}$$

*holds $\mathcal{L}(G_i, \beta) \neq \mathcal{L}(G_i, \beta_j)$.*

**No identical parameters:** *For all $\beta : X_i \to \mathcal{T}_\Sigma$ and all $x_i, x_j \in X_i$ holds: For all unfoldings $\upsilon_w \in \Upsilon_i, w \in W$ (see def. 7.29) with instantiation $\beta_w$ of variables follows $i = j$ from $\beta_w(x_i) = \beta_w(x_j)$.*

Note, that similar restrictions were formulated by (Le Blanc, 1994). To sum up the given criteria for minimality: An RPS is minimal if it only contains recursive equations which are called at least once, if each parameter in the head of an equation is used at least once for instantiating a parameter in the body of the equation, and if there exist no parameters with different names but (always) identical instantiation. It is obvious, that each RPS which does not fulfill these criteria can be transformed into one which does fulfill them by deleting unused equations and parameters and by unifying parameters with different names but identical instantiations. That is, these criteria do not restrict the expressiveness of RPSs.

A more strict definition of minimality could additional consider the size (i. e., number of symbols) of the subprogram bodies (Osherson, Stob, and Weinstein, 1986). But this makes only sense for RPSs with a single recursive equations. In that case, minimality of a subprogram $G$ is given if there exists no subprogram $G'$ with $\mathcal{L}(G', \beta) \subset \mathcal{L}(G, \beta)$ (Mühlpfordt and Schmid, 1998). For RPSs with more than one equation, the minimality of each single equation cannot be determined by comparison of languages: If equation $G$ calls other equations $G_i$ there is always a smaller language where these calls are replaced by constant symbols. Therefore, it would be necessary to define minimality over the size of all equation bodies and the term representing the main program. There are too

many degrees of freedom to define a thight criterium: For example, is an RPS consisting of a small main program but complex subprograms smaller than an RPS consisting of a complex main program and small subprograms?

To guarantee uniqueness for calculating the substitutions in a subprogram body for a given initial program, we introduce the notion of "substitution uniqueness":

**Definition 7.33 (Substitution Uniqueness of an RPS)** *Let $T_{init} \subset \mathcal{T}_{\Sigma\cup\{\Omega\}}$ be a set of initial programs and $\mathcal{S} = (\mathcal{G}, t_0)$ an RPS over $\Sigma$ and $\Phi$ which explains $T_{init}$ recursively (see def. 7.30). $\mathcal{S}$ is called substitution unique with regard to $T_{init}$ if there exists no $\mathcal{S}'$ over $\Sigma$ and $\Phi$ which explains $T_{init}$ recursively and for which holds:*

- $t'_0 = t_0$,
- *for all $G_i \in \Phi$ holds $\mathbf{pos}(t_i, G_i) = \mathbf{pos}(t'_i, G_i) = U_{rec}$, $t'_i[U_{rec} \leftarrow \Omega] = t_i[U_{rec} \leftarrow \Omega]$, and it exists an $r \in R$ with $\mathbf{sub}(x_j, r) \neq \mathbf{sub}'(x_j, r)$ (see def. 7.27).*

Substitution uniqueness guarantees that it is not possible to replace a substitution term in $\mathcal{S}$ such that the resulting RPS $\mathcal{S}'$ still explains a given set of initial programs recursively.

## 2.3 THE SYNTHESIS PROBLEM

Now all preliminaries are given to state the synthesis problem:

**Definition 7.34 (Synthesis Problem)** *Let $T_{init} \subset \mathcal{T}_{\Sigma\cup\{\Omega\}}$ be a set of initial programs with indices $I = \{1, \ldots, |T_{init}|\}$. The synthesis problem is to induce*

- *a signature $\Sigma$,*
- *a set of function variables $\Phi = \{G_1, \ldots, G_n\}$,*
- *a minimal RPS $\mathcal{S} = (\mathcal{G}, t_0)$ with a main program $t_0 \in \mathcal{T}_{\Sigma\cup\Phi}(X)$ and a set of recursive equations $\mathcal{G} = \langle G_1(x_1, \ldots, x_{m_1}) = t_1, \ldots, G_n(x_1, \ldots, x_{m_n}) = t_n\rangle$*

*such that*

- *$\mathcal{S}$ recursively explains $T_{init}$ (def. 7.30), and*
- *$\mathcal{S}$ is substitution unique (def. 7.33).*

In the following section, an algorithm for solving the synthesis problem will be presented. Identification of a signature $\Sigma$ is dealt with only implicitly – the RPS is constructed exactly over such symbols which are necessary for explaining the initial programs recursively. Because the folding mechanism should be

independent of the way, in which the initial programs were constructed (by a planner, as input of a system user), we allow for incomplete unfoldings and we allow instantiated initial programs as well as programs over parameters (i. e., generic traces), where parameters are treated just like constants.

If synthesis starts with only a single initial program, the main program can contain no parameters (except the parameters which where identified as "constants"). If a parameterized main program is searched for (which is the usual case), we assume that all parameters inferred for the subprograms which are called from the main program are also parameters of the main program itself.

# 3 SOLVING THE SYNTHESIS PROBLEM

In this section, the algorithms for constructing an RPS from a set of initial programs (trees) are presented: For each given initial tree, in a first step, a segmentation is constructed (sect. 3.1), which corresponds to the unfoldings (see def. 7.29) of a hypothetical subprogram. As intermediate steps, first a set of hypothetical recursion points (see def. 7.27) is identified and a skeleton of the body of the searched for subprogram is constructed.

If recursion points do not explain the complete initial tree, all unexplained subtrees are considered as a new set of initial trees for which the synthesis algorithm is called recursively and the subtrees are replaced by the name $G_i$ of the to be inferred subprogram (sect. 3.3). If an initial tree cannot be segmented starting from the root, a constant initial part is identified and included in the body of the calling function – that is, in $t_0$ for top-level subprograms and in $G_i$ if $\mathbf{calls}(G_i, G_j)$ and $G_j$ is the currently considered hypothetical subprogram (see figure 7.13 in section 3.5.2).

For a given segmentation, the maximal pattern is constructed by including all parts of initial trees which are constant over all segments into the subprogram body (sect. 3.2). All remaining parts of the segments are considered as parameters and their substitutions. That is, in a last step, a unique substitution rule must be calculated and as a consequence, the parameter instantiations of the function call are identified (sect. 3.4). The only backtrack-point for folding is calculating a valid segmentation for the initial trees.

We begin each subsection with an illustration. For readers not interested in the formal details, it is enough to study the illustration for getting the general idea of our approach.

## 3.1 CONSTRUCTING SEGMENTATIONS

In the following, we first give an informal illustration. Then, we introduce some concepts for construction segmentations of initial programs. Afterwards,

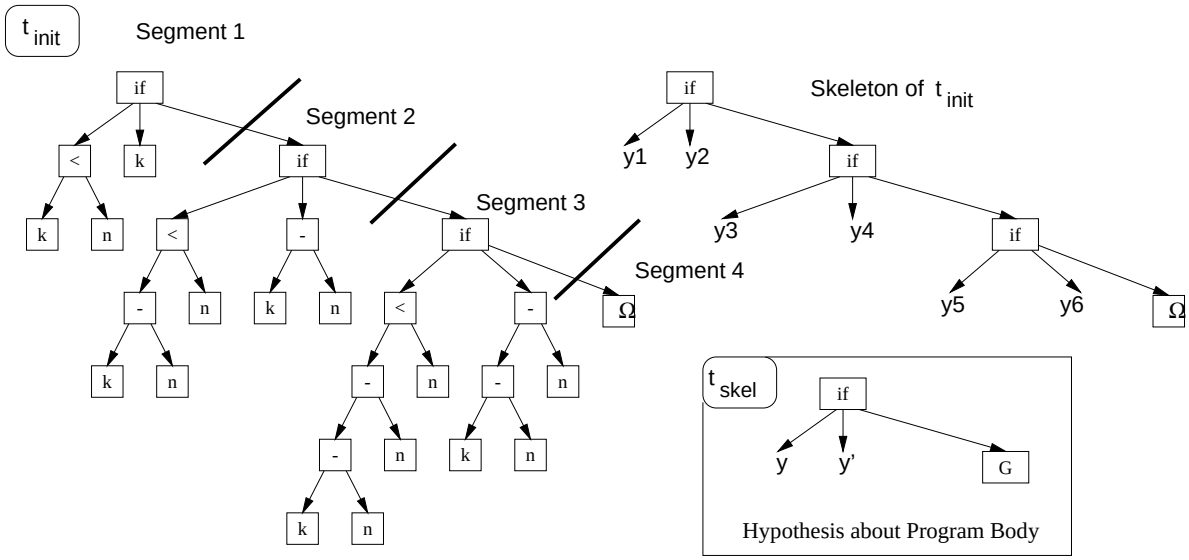



*Figure 7.6.* Valid Recurrent Segmentation of *Mod*

we introduce criteria which must hold for a segmentation to be valid. Finally, we present an algorithm for constructing valid segmentations.

### 3.1.1 SEGMENTATION FOR 'MOD'

Consider the simple RPS for calculating *Mod(k, n)*, which consists of a single recursive equation and where the main program $t_0$ is just the call of this equation:

$\mathcal{S} = \langle \mathcal{G}, t_0 \rangle$

$\mathcal{G} = \langle$ Mod(k, n) = if(<(k, n), k, Mod(−(k, n), n)) $\rangle$

$t_0 = Mod(k, n)$

Remember, that all recursive equations are considered as "subprograms". The main program, in general, can be a term which calls one or more of the recursive sub-programs of the RPS (see sect. 1.3.1).

In figure 7.6 a term is given which can be folded into this RPS. For better readability, we do not give an initial instantiation for parameters $k$ and $n$. The first and crucial step of folding is, to identify a *valid recurrent segmentation* from an initial program term $t_{init}$. For the given term, such a segmentation is marked in the figure.

As a first step, all paths in the term leading to Ωs are identified; that is, paths to positions where the unfolding of the hypothetical (searched for) RPS was stopped. These paths constitute the *skeleton* of the term, which is defined as

the minimal pattern of the term (see def. 7.11) which contains all Ωs and where all sub-terms which are not on paths to Ωs are ignored.

Then, a set of hypothetical recursion points $U_rec$ (see def. 7.27) is generated. There are different possible strategies, to obtain such a set from $t_{init}$. The most simple hypothesis is, that, beginning at the root, each node in the skeleton constitutes an unfolding position. For our example, this hypothesis is $U_{rec} = \{3.\lambda\}$. We can construct hypothetical unfolding indices $W$ from $U_{rec}$ (see def. 7.28) which leads to the segmentation given in figure 7.5.

Replacing all subtrees at positions $U_{rec}$ in the skeleton by Ωs results in a hypothesis about the program body $t_{skel}$. For our example, the segmentation induced by $U_{rec} = \{3.\lambda\}$ is valid because each segment consists of the same (non-trivial) pattern and because all Ωs (here only a single one) can be reached and this Ω has a position which corresponds to a recursion point (unfolding position). The valid segmentation is recurrent because the skeleton of the upper-most segment, $t_{skel}$ constitutes a non-trivial pattern for all subtrees $t_u$ of the initial program where $u$ is a hypothetical recursion point. The notion of a valid recurrent segmentation follows directly from the definition of a recursive explanation of a recursive program (see def. 7.30).

If a valid recurrent segmentation can be found, the result of this first step of folding is a skeleton of the sub-program body. For our example it is *if(y, y', G)*, where *G* is a new function variable in Φ.

As stated in the definition of the synthesis problem (see def. 7.34), input into the folding algorithm is a *set* of initial programs $T_{init}$. That is, a user or another system can provide more than one example program which are considered as unfoldings of the same searched-for RPS with different instantiations and/or different unfolding depth. In our example, $T_{init}$ consisted of a single initial program. For some definitions below, it is enough, to consider a single program $t_{init} \in T_{init}$, for others, it is important, that the complete set is taken into account.

In general, finding a valid recurrent segmentation can be much more complicated as illustrated for *Mod*:

- It might not be possible to find a valid recurrent segmentation starting at the root of the initial tree, that is, there exists a constant initial part of the term which belongs to the main program $t_0$. In that case, this initial part is introduced in $t_0$ and segmentation starts with $T_{init}$ as set of all sub-terms of this initial part.

- The recurrence relation underlying the term can be of arbitrary complexity. That is, in general, the skeleton does not just exist of a single path. Our strategy for constructing segmentations is to find segmentations which start as high in the given tree as possible and which cover as large a part of

the tree as possible. As we will see in the algorithm below, we search for hypothetical recursion points "from left to right" in the tree.

- There might exist sub-terms in the given initial program which contain $\Omega$s but which cannot be covered within any valid recurrent segmentation. In this case, it is assumed, that these subtrees belong to calls of a further recursive equation. The positions of such subtrees are considered as "sub-schema positions" $U_{sub}$ and finding valid recursive segmentations for these subtrees is dealt with by starting the folding algorithm again with these subtrees as new set of initial trees.

- We allow for incomplete unfoldings. That is, some paths in the initial program are truncated before an unfolding is complete. For example, for the recursive equation

  *addlist(l) = if(empty(l), 0, +(head(l), addlist(tail(l))))*

  the last unfolding could result in an $\Omega$ at the position of $+$, because the *else* case is not considered if list $l$ is empty. Although we consider unfolding as a purely syntactical process, it might be useful to take into account that construction of the initial program was based on some kind of semantic information (such as evaluation of expressions).
  Consequently, when constructing a segmentation, it is allowed that a segment contains an $\Omega$ above the hypothetical unfolding position (but not below!).

All these cases are covered in our approach which we will introduce now more formally.

### 3.1.2 SEGMENTATION AND SKELETON

The first and crucial step for inducing an RPS from a set of initial programs is the identification of a valid, recurrent segmentations of the initial trees. When searching for a segmentation, at first only such nodes of the initial trees are considered which are lying on paths leading to hypothetical recursion points. These paths constitute a special (minimal) pattern of an initial tree, called skeleton:

**Definition 7.35 (Skeleton)** *The skeleton of a term $t \in \mathcal{T}_{\Sigma\cup\{\Omega\}}(X)$, written $\mathbf{skeleton}(t)$ is the minimal pattern (def. 7.11) of t for which holds $\mathbf{pos}(t,\Omega) = \mathbf{pos}(\mathbf{skeleton}(t),\Omega)$.*
*Let $U \subseteq \mathbf{pos}(t,\Omega)$ in $t$ with $t_U = \Omega$ for all $u \in U$. Then we define $\mathbf{skeleton}(t,U)$ as that skeleton of term $t$ which contains only the $\Omega$s at positions $u \in U$, that is, the minimal pattern with $\mathbf{pos}(\mathbf{skeleton}(t,U),\Omega) = U$.*

An example skeleton for *Mod* is given in figure 7.5. This is the simple case of a single initial program which can be explained by an RPS consisting of a single recursive equation and a main program which just calls this equation. Note, that the paths under consideration are leading to $\Omega$s, that is, truncation points of the unfolding of a hypothetical RPS.[3]

In general, it can be necessary for finding a recursive explanation of an initial program to identify a subprogram $G_i$ which calls further subprograms. The calls of these subprograms must be at fixed positions in $G_i$, called sub-schema positions[4]. In that case, the skeleton of $G_i$ does not contain all paths leading to $\Omega$s (see def. 7.35). For these remaining $\Omega$s must hold, that they can be reached over paths from the hypothetical sub-schema positions. Recursion points and sub-schema positions constitute a *hypothesis for inducing an RPS.*

**Definition 7.36 (Recursion Points and Sub-Schema Positions)** *Let $U_{rec}$ and $U_{sub}$ be two sets of positions with $U_{rec} \neq \emptyset$. If for all $u_{sub} \in U_{sub}$ and all positions $u$ above $u_{sub}$ ($u_{sub} = u \circ i, i \in N$) holds*

1. $\nexists u'_{rec} \in U_{rec}$ *with* $u_{sub} \leq u'_{rec}$ *and*
2. $\exists u_{rec} \in U_{rec}$ *with* $u \leq u_{rec}$,

*then $(U_{rec}, U_{sub})$ is a hypothesis over recursion points $U_{rec}$ and sub-schema positions $U_{sub}$.*

For the initial program given in figure 7.7 (see tab. 7.2 for the underlying RPS) the recursion point is $3.2.\lambda$, that is the second "*if*" on the rightmost path. The sub-schema position is $3.1.\lambda$, that is, the "*if*" in the left path under "*cons*". For this position holds that it is not above the recursion point (condition 1 of def. 7.36) but it can be reached over a path leading to this recursion point (condition 2 of def. 7.36): $u_{sub} = u \circ 1 = 3.1.\lambda$ for $u = 3.\lambda$.

### 3.1.3 VALID HYPOTHESES

Recursion points and sub-schema positions constitute a valid hypothesis if they are on paths leading to $\Omega$s and where the $\Omega$s are at positions corresponding to recursion points or – for incomplete unfoldings – above recursion points. For example, in the initial tree given in figure 7.7 the last unfolding is incomplete: After the 'if'-node in the right-most path should follow another 'cons'-node; but, instead it follows an $\Omega$.

[3]For initial trees generated by a planner or by a system user that means, that cases for which the continuation is unknown or not calculated must be marked by a special symbol.

[4]Because the inference of a further sub-program again can involve a constant initial part, not just a recursive equation but an additional "main" could be inferred which will then be integrated in the calling subprogram. Therefore, we speak of sub-schema positions rather than subprogram positions

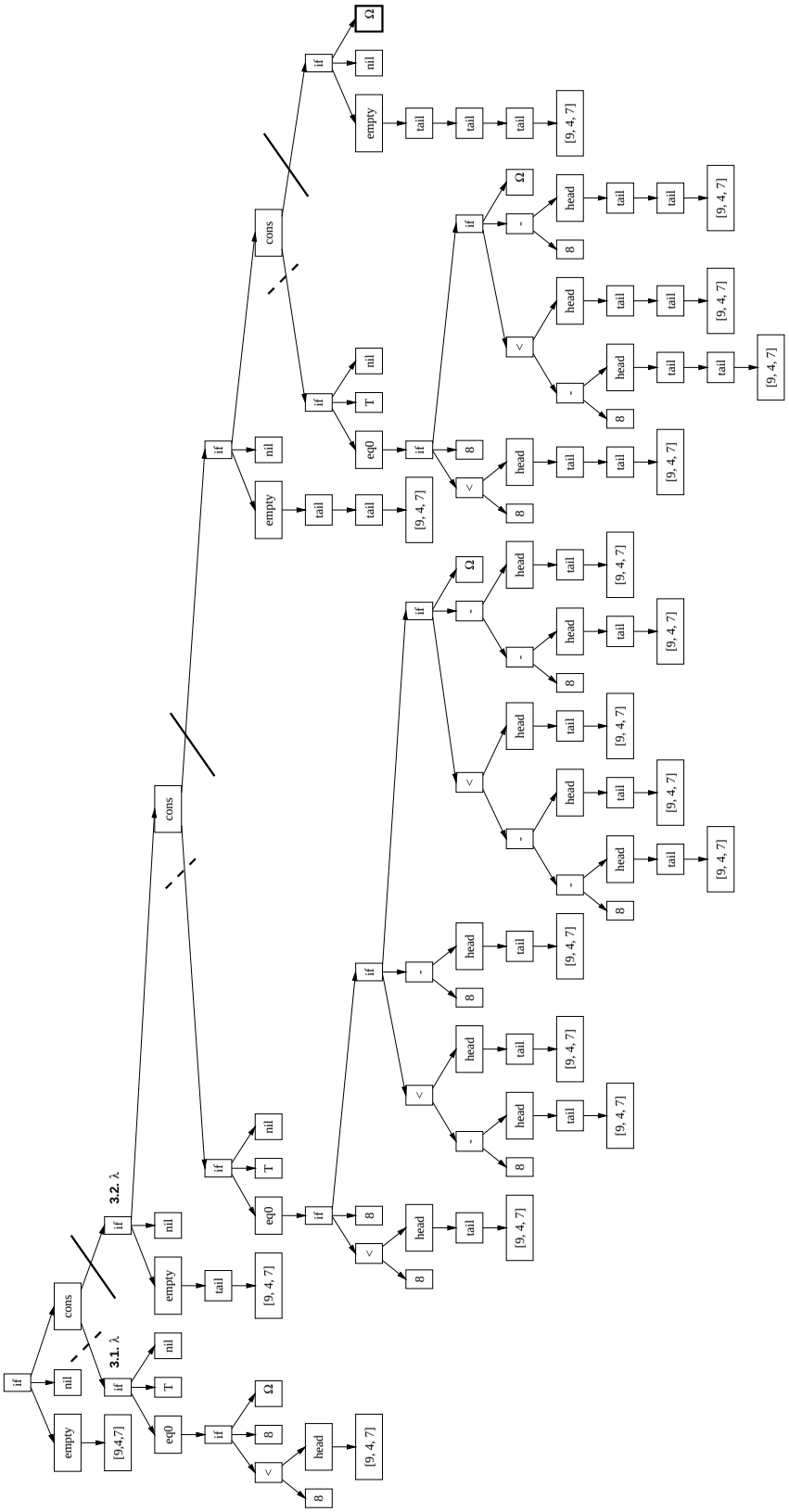



*Figure 7.7.* Initial Program for *ModList*

**Definition 7.37 (Valid Recursion Points and Sub-Schema Positions)** *Let be $t_{init}$ an initial tree and $(U_{rec}, U_{sub})$ a hypothesis. $(U_{rec}, U_{sub})$ is a valid hypothesis for $t_{init}$ if*

1. $\forall u \in U_{sub}$*: If* $u \in \mathbf{pos}(t_{init})$*, then* $\mathbf{pos}(t_{init}|_u, \Omega) \neq \emptyset$*,*
2. $\forall u \in U_{rec}$*:* $u \in \mathbf{pos}(t_{init})$ *and* $(U_{rec}, U_{sub})$ *is valid for* $t_{init}|_u$ *or*
3. $\forall u \in U_{rec}$*:* $\exists u_\Omega \in t_{init}$ *with* $u_\Omega < u$ *and* $t_{init}|_{u_\Omega} = \Omega$*.*

The first condition of definition 7.37 ensures that sub-schema positions are at such positions in the initial tree where the subtrees contain at least one $\Omega$. Condition 2 recursively ensures that the hypothesis holds for all sub-terms at the recursion points. Condition 3 ensures that the $\Omega$s have positions at or above recursion points.

Up to now we have only regarded the *structure* of an initial tree. A valid segmentation additionally must consider the symbols of the initial tree (i. e., the node labels). Therefore, hypothesis $(U_{rec}, U_{sub})$ will be extended to a term $t_{skel}$ which represents a hypothesis about a sub-program body.

**Definition 7.38 (Valid Segmentation)** *Let* $t_{init} \in \mathcal{T}_{\Sigma \cup \{\Omega\}}$ *with* $\mathbf{pos}(t, \Omega) \neq \emptyset$ *be an initial program. Let* $(U_{rec}, U_{sub})$ *be a valid hypothesis about recursion points and sub-schema positions in t. Let* $t_{skel} \in \mathcal{T}_{\Sigma \cup \{\Omega\}}(X)$ *be a term with* $t_{skel} \leq_\Omega t_{init}$ *(def. 7.26) and* $U_{rec} \cup U_{sub} = \mathbf{pos}(t_{skel}, \Omega)$*. A segmentation given by* $(U_{rec}, U_{sub})$ *together with* $t_{skel}$ *is valid for* $t_{init}$ *if for*

- $U_c = U_{rec} \cap \mathbf{pos}(t_{init})$ *(the set of recursion points in* $t_{init}$*), and*
- $U_{ic} = U_{rec} \setminus U_c$ *(the set of recursion points not in* $t_{init}$*)*

*holds:*

**Reachability of $\Omega$s:** *For all* $u_{ic} \in U_{ic}$ *exists a* $u_\Omega \in \mathbf{pos}(t_{init}, \Omega)$ *with* $u_\Omega < u_{ic}$*.*

**Validity of the Skeleton:** *Given the set of all positions of* $\Omega$*s in* $t_{init}$ *which are above the recursion points as* $U^t_{ic}\{u_\Omega \mid u_\Omega < u, u_\Omega \in \mathbf{pos}(t_{init}, \Omega), u \in U_{ic}\}$ *it holds* $t_{skel}[U^t_{ic} \leftarrow \Omega] \leq_\Omega t_{init}$*.*

**Validity of Segmentation in Subtrees:** *For all* $u \in U_c$ *holds* $(U_{rec}, U_{sub})$ *together with* $t_{skel}$ *is a valid segmentation for* $t|_u$*.*

The definition is somewhat complicated because we are allowing incomplete unfoldings. Note, that we currently are *not* concerned with the question of how recursion points are obtained. We just assume that a non-empty set of hypothetical recursion points $U_{rec}$ is at hand which can contain positions which can be found in the given $t_{init}$ and can contain positions which are not in $t_{init}$.

The term $t_{skel}$ is a hypothesis about the skeleton of the body of a searched for subprogram. It contains $\Omega$s (truncations of unfoldings) at the hypothetical recursion points and sub-schema positions. For incomplete unfoldings, for each given recursion point there must exist an $\Omega$ at or *above* this point ("reachability"). The set $U_{ic}^t$ contains all such positions and the term $t_{skel}[U_{ic}^t \leftarrow \Omega]$ is the skeleton of the hypothetical subprogram body reduced to the size of the currently investigated and possibly incomplete sub-term. For a valid hypothesis it must hold further, that the currently investigated term contains a sub-term with at least one $\Omega$ and that there are no further $\Omega$s in $t_{init}$. The positions in $U_{sub}$ cover all $\Omega$s which are not generated by the currently investigated subprogram ("validity of skeleton"). Finally, these conditions must hold for all further hypothetical segments ("validity of segmentation in subtrees"). Note, that for a tree $t_u$, which is a subtree of $t_{init}$ with a recursion point as root, it can happen that not all positions in $U_{rec}$ are given!

If we extend this definition from one initial tree $t_{init}$ to a set of initial trees $T_{init}$, we request that at least one of the initial programs contains a completely unfolded segment:

**Definition 7.39 (Segmentation of a Set of Initial Programs)** *$(U_{rec}, U_{sub})$ together with $t_{skel}$ is a valid segmentation of a set of initial programs $T_{init}$ if there exists a $t_{init} \in T_{init}$ with $t_{skel} \leq_\Omega t_{init}$ and if $(U_{rec}, U_{sub})$ together with $t_{skel}$ is a valid segmentation of all $t \in T_{init}$ as defined in 7.38.*

A valid segmentation partitions an initial tree into a sequence of segments. These segments can be indexed in analogy to the unfoldings of a subprogram (def. 7.29):

**Definition 7.40 (Segments of an Initial Program)** *Let $T_{init} \subseteq \mathcal{T}_{\Sigma\cup\{\Omega\}}$ be a set of initial programs and $E$ a set of indices with $t_e \in T_{init}$ for all $e \in E$. Let $R = \{1, \ldots, |U_{rec}|\}$ be a set of indices of recursion points. Let $(U_{rec}, U_{sub})$ together with $t_{skel}$ be a valid segmentation for all $t_e \in T_{init}$ and $W$ the set of indices over segments constructed over $U_{rec}$. Let $G$ be a new function variable with $G \notin \Sigma$. Then we can define $\mathbf{segment}(t_e, U_{rec}, w)$ with $G$ as name for the subprogram as segment of $t_e$ with respect to recursion points $U_{rec}$ with segment index $w \in W$ inductively:*

$$1.\ \mathbf{segment}(t, U_{rec}, \lambda) = \begin{cases} t[U_{rec} \cap \mathbf{pos}(t) \leftarrow G] & if\ t \neq \Omega \\ \bot & otherwise \end{cases}$$

$$2.\ \mathbf{segment}(t, U_{rec}, r.v) = \begin{cases} \mathbf{segment}(t|_{u_r}, U_{rec}, v) & if\ r \in R\ and\ u_r \in \mathbf{pos}(t) \\ \bot & otherwise. \end{cases}$$

Function $\mathbf{skeleton}(t_e, U_{rec}, w)$ returns for a given initial program $t_e$ and a set of hypothetical recursion points $U_{rec}$ the according segment with index

$w$ if it is contained in the tree and otherwise "unknown".[5] A new function variable is inserted at the position of the recursion point which will be the name of the to be induced subprogram. For the case of incomplete unfoldings the $\Omega$s positioned above hypothetical recursion points remain in the tree. It is not possible, that a hypothetical subprogram body can consist of the subprogram name $G$ only (item 1 in def. 7.40).

The segments of an initial program correspond to unfoldings of a hypothetical recursive equation as defined in 7.29. But (up to now), segments can additionally contain subtrees (unfoldings) of further recursive equations (see sect. 3.3).

The set of all indices for a given initial program with given segmentation can be defined as

**Definition 7.41 (Segment Indices)** *Let $T$ be a set of initial trees indexed over $E$ with $t_e \in T$ for all $e \in E$. Let $(U_{rec}, U_{sub})$ together with $t_{skel}$ be a valid segmentation for all terms in $T$ and let $R = \{1, \ldots, |U_{rec}|\}$ be a set of indices of recursion points and $W$ the set of unfolding indices calculated over $R$ (def. 7.28). The $W(e)$ is the set of indices over the segments contained in an initial tree $t_e$ with $W(e) = \{w \mid w \in W, \mathbf{segment}(t_e, U_{rec}, w) \neq \bot\}$.*

For inducing a recursive subprogram from a given initial program, this initial program must be of "sufficient size", that is, it must contain a sufficient number of (hypothetical) unfoldings such that all information necessary for folding can be obtained from the given tree. Especially, it is necessary that each hypothetical subprogram is unfolded at least once at each hypothetical recursion point. A hypothesis over recursion points of a subprogram can only lead to successful folding if the following proposition holds:

**Lemma 7.1 (Recurrence of a Valid Segmentation)**
*Let $(U_{rec}, U_{sub})$ together with $t_{skel}$ be a valid segmentation for a set of initial programs $T_{init}$ and let $R = \{1, \ldots, |U_{rec}|\}$ be the set of indices over recursion points $U_{rec}$. $(U_{rec}, U_{sub})$ can only generate a subprogram which recursively explains $T_{init}$, if for each recursion point $u_r \in U_{rec}$ there exists an initial tree $t_e \in T_{init}$ such that exists $w \in W(e)$ with $w = v \circ r, v \in W(e), r \in R$.*
*Proof: follows directly from definition 7.30.*

Lemma 7.1 is a weak consequence from the definition of recursive explanations (def. 7.30), because it is not required that segments correspond to complete unfoldings. That is, we have given just a necessary but not a sufficient condition for successful induction.

[5] Note, that we use the symbol $\bot$ to represent an unknown segment rather than $\Omega$ to discriminate between an undefined sub-term in a term ($\Omega$) and an undefined hypothetical function body ($\bot$).

### 3.1.4 ALGORITHM

The definition of a valid segmentation (def. 7.38) allows us to formulate some relations which can be used for an algorithmic construction of a segmentation hypothesis:

- Because of conditions $t_{skel} \leq_\Omega t_{init}$ and $U_{rec} \subseteq \mathbf{pos}(t_{skel})$ must hold $U_{rec} \subseteq \mathbf{pos}(t_{init})$.

- From the condition "validity of segmentation in subtrees" follows for all $u_{rec} \in U_{rec}$ that $\mathbf{node}(t_{init}, u_{rec}) = \mathbf{node}(t_{init}, \lambda)$. That is, only such positions in an initial tree can be candidates for recursion points which contain the same symbol as its root node.[6]

- The positions of further sub-schemata can be inferred from the uppermost unfolding in $t_init$: $U_{sub} = \{u \mid u \in \mathbf{lpos}(\mathbf{skeleton}(t_{init}[U_{rec} \leftarrow \Omega])) \setminus U_{rec}, \mathbf{pos}(t_{init}|_u, \Omega) \neq \emptyset\}$. (Remember that $\mathbf{lpos}(t)$ returns the positions of leaf nodes of a term, see def. 7.6).

- Consequently, a hypothesis $t_{skel}$ about the skeleton of the subprogram body can be constructed: $t_{skel} = \mathbf{skeleton}(t_{init}[(U_{rec} \cup U_{sub}) \leftarrow \Omega])$.

As mentioned before, constructing a hypothesis about the set of possible recursion points $U_{rec}$ from a given set of initial trees is the first and crucial step of inducing an RPS. This hypothesis determines the set of sub-schema positions and the skeleton of a searched for subprogram body. It represents an assumption $(U_{rec}, U_{sub})$ about the segmentation of an initial tree which corresponds to the $i$-th unfolding of a searched for recursive equation. The validity of a hypothesis can be initially checked using the criteria given in definition 7.38 and it can checked whether it holds recursively over an complete initial tree using lemma 7.1. If validation fails, a new set of recursion points $U_{rec}$ must be constructed. If no such set can be found, the searched for RPS does not consist of a main program which just calls a recursive equation, and search for recursion points must start at deeper levels of the tree. In the worst case, the complete tree would be regarded as main program with an empty set of recursive equations.[7]

For constructing possible sets of recursion points, each given initial tree must be completely traversed. The construction is based on a *strategy* which determines the sequence in which possible hypotheses are generated. The

[6] Note, that it is possible, that a given initial tree contains a constant part belonging to the body of the calling function, such that the initial tree for which a recursive subprogram is to be induced has a root which is on a deeper position.

[7] Because we search for an RPS with at least one recursive equation, our algorithm terminates already at an earlier point – if the remaining tree is so small, that it is not possible to find at least one unfolding for each hypothetical recursion point.

*Table 7.6.* Calculation of the Next Position on the Right

**Function:** $\mathbf{nextPosRight} : \mathcal{T}_{\Sigma\cup\{\Omega\}} \times \mathbf{pos}(t_{init}) \rightarrow \mathbf{pos}(t_{init})$ for all $t_{init} \in T_{init}$
**Pre:** $T_{init}$ is a set of initial trees; $u$ is a position in the initial trees
**Post:** next position right of $u$ if such a position exists, $\perp$ otherwise

1. **IF** $u = \lambda$
2. **THEN return** $\perp$
3. **ELSE**
   (a) Let $u = u' \circ k$
   (b) **IF** $\exists t \in T_{init}$ with $u' \circ (k+1) \in \mathbf{pos}(t)$
   (c) **THEN return** $u' \circ (k+1)$
   (d) **ELSE return** $\mathbf{nextPosRight}(T_{init}, u')$.

proposed strategy ensures that minimal subprograms which explain maximally large parts of an initial tree are constructed:

Search for a segmentation, starting with $U_{rec} = \emptyset$ at position $u = 1.\lambda$ considering the following cases:

**Case 1:** (The initial trees are completely traversed.)
If there exists no further position

- If $U_{rec} \neq \emptyset$ then construct $U_{sub}$ and $t_{skel}$ with respect to $U_{rec}$.
  If $(U_{rec}, U_{sub})$ together with $t_{skel}$ is a valid segmentation, then *stop* else *backtrack*.
- If $U_{rec} = \emptyset$ then no recursion points could be found and consequently, there exists no subprogram which recursively explains the initial trees.

**Case 2:** (The node at position $u$ is recursion point.)
Set $U_{rec} = U_{rec} \cup \{u\}$.
Construct $U_{sub}$ and $t_{skel}$ with respect to $U_{rec}$.
If $(U_{rec}, U_{sub})$ together with $t_{skel}$ is a valid segmentation, then progress with $U_{rec}$ to the next position $u'$ *right* of $u$.
Otherwise go to case 3.

**Case 3:** (The node at position $u$ is lying above a recursion point.)
If there exists a position $u' = u \circ 1$ in the initial trees, progress with $U_{rec}$ at position $u'$.
Otherwise go to case 4.

**Case 4:** (There are no nodes lying below the current node.)
Progress with $U_{rec}$ at the next position $u'$ *right* of $u$.

This strategy realizes a search for recursion points from left to right where for each hypothetical recursion point search proceeds downward. The function for calculating the next position to the right is given in table 7.6.

*Table 7.7.* *Factorial* and Its Third Unfolding with Instantiation $succ(succ(0))$ (a) and $pred(3)$ (b)

**Subprogram:** *G(x) = if(eq0(x), 1, *(x, G(pred(x))))*
**Unfolding (a):** Third unfolding for $\beta(x) =$ *succ(succ(0))*

$t_{init} =$ if(eq0(succ(succ(0))), 1, *(succ(succ(0)),
if(eq0(pred(succ(succ(0)))), 1, *(pred(succ(succ(0))),
if(eq0(pred(pred(succ(succ(0))))), 1, *(pred(pred(succ(succ(0)))),Ω))))))

**Unfolding (b):** Third unfolding for $\beta(x) =$ *pred(3)*

$t_{init} =$ if(eq0(pred(3)), 1, *(pred(3),
if(eq0(pred(pred(3))), 1, *(pred(pred(3)),
if(eq0(pred(pred(pred(3)))), 1, *(pred(pred(pred(3))),Ω))))))

## 3.2 CONSTRUCTING A PROGRAM BODY

For a given valid segmentation the body of a hypothetical subprogram can be constructed. For each node of a segment it must be decided whether it belongs to

- the body of the subprogram, or
- the parameter instantiation, or
- a further subprogram.

In this section we will only consider subprograms without calls of further subprograms, that is $U_{sub} = \emptyset$. An extension for subprograms with additional subprogram calls is given in section 3.3. In the following, we will first give an example for constructing the program body. Afterwards, we will introduce a theorems concerning the maximization of the program body. Finally, we will present an algorithm for calculating the program body for a given segmentation of a set of initial trees.

### 3.2.1 SEPARATING PROGRAM BODY AND INSTANTIATED PARAMETERS

For illustration we use a simple linear subprogram – the recursive equation for calculating the *factorial* of a natural number. This subprogram and its third unfolding is given in table 7.7. The parameter $x$ is instantiated with 2. In the first case, represented as *succ(succ(0))* and in the second case, represented as *pred(3)* (short for *pred(succ(succ(succ(0))))*).

*Table 7.8.* Segments of the Third Unfolding of *Factorial* for Instantiation *succ(succ(0))* (a) and $pred(3)$ (b)

| (a) Index $w$ | Segment | | |
|---|---|---|---|
| $\lambda$ | if(eq0(succ(succ(0))), | 1, | *(succ(succ(0)), Ω)) |
| $1.\lambda$ | if(eq0(pred(succ(succ(0)))), | 1, | *(pred(succ(succ(0))), Ω)) |
| $1.1.\lambda$ | if(eq0(pred(pred(succ(succ(0))))), | 1, | *(pred(pred(succ(succ(0)))),Ω)) |

| (b) Index $w$ | Segment | | |
|---|---|---|---|
| $\lambda$ | if(eq0(pred(3)), | 1, | *(pred(3), Ω)) |
| $1.\lambda$ | if(eq0(pred(pred(3))), | 1, | *(pred(pred(3)), Ω)) |
| $1.1.\lambda$ | if(eq0(pred(pred(pred(3)))), | 1, | *(pred(pred(pred(3))),Ω)) |

A valid recurrent segmentation for $t_{init}$ is $U_{rec} = \{3.2.\lambda\}$, and the skeleton is $t_{skel} = if(y_1, y_2, *(y_3, \Omega))$. The segments for $t_{init}$ are given in table 7.8.

The program body can be constructed as maximal pattern (see def. 7.13) of all given segments. For the segments constructed from the initial tree with instantiation $\beta(x) =$ *succ(succ(0))*, the program body therefore is

$$t_{G'} = if(eq0(x'), 1, *(x', G')).$$

The body is identical with the definition of *factorial* given in table 7.7.

For the segments constructed from the initial tree with instantiation $\beta(x) =$ *pred(3)*, the program body is

$$t_{G''} = if(eq0(pred(x'')), 1, *(pred(x''), G')).$$

A part of the parameter instantiation got part of the program body! This is a consequence from defining the program body as maximal pattern of the segments of the initial trees. If an instantiation is given which shares a non-trivial pattern (i. e., the anti-instance is not just a variable; see def. 7.11) with the substitution term (in the recursive call), then this pattern will be assumed to be part of the program body. A simple practical solution is to present the folding algorithm with a set of initial trees with different instantiations.

In the following, it will be shown that if a recursive equation exists for a set of initial trees then we can find a recursive subprogram with a "maximized" body. This resulting subprogram together with the found parameter instantiation is equivalent to the "intended" subprogram with the "intended" instantiation. Of course, it does *not* hold, that the induced program and the intended program are equivalent for *all possible* parameter instantiations!

### 3.2.2 CONSTRUCTION OF A MAXIMAL PATTERN

The body of a subprogram is constructed by including all nodes in the skeleton which are common over all segmentations. All subtrees which differ over

segments are considered as instantiated parameters, that is, the initial instantiations of the parameters when the subprogram is called and the instantiations resulting from substitutions of parameters in recursive calls. A given set of initial trees – in the following called examples – can contain trees with different initial parameter instantiations as shown in the *factorial* example above.

**Theorem 7.1 (Maximization of the Body)**
*For a set of example initial trees, let $E$ be a set of examples indices $e$ with $\mid E \mid \geq 1$. For each recursive subprogram $G(x_1, \ldots, x_n) = t_G$ with $X = \{x_1, \ldots, x_n\}$ and $t_G \in \mathcal{T}_{\Sigma \cup \Phi}(X)$ together with initial instantiations $\beta_e : X \to \mathcal{T}_\Sigma$ for all $x \in X$ and all $e \in E$ exists a subprogram $G'(x_1, \ldots, x_{n'}) = t_{G'}$ with $X' = \{x_1, \ldots, x_{n'}\}$ and $t_{G'} \in \mathcal{T}_{\Sigma \cup \Phi}(X')$ together with initial instantiations $\beta'_e : X' \to \mathcal{T}_\Sigma$ for all $x \in X'$ and all examples $e \in E$ such that $\mathcal{L}(G, \beta_e) = \mathcal{L}(G', \beta'_e)$ for all $e \in E$. Additionally, for each $x \in X'$ holds that the instantiations which can be generated by $G'$ from $\beta'_e$ don't share a common not-trivial pattern.*
*Proof: see appendix B2.*

Theorem 7.1 states that *if* a recursive subprogram $G$ exists for a set of initial trees $T_{init} = \{t_e \mid e \in E\}$ which can generate all $t_e \in T_{init}$ for a given initial instantiation $\beta_e$, *then* there also exists a subprogram $G'$ such that the instantiations (over all segments and all examples) do not share a non-trivial pattern (a common prefix). Therefore, it is feasible to generate a hypothesis about the subprogram body for a given segmentation with recursion points $U_{rec}$ by calculating the *maximal* pattern of the segments.

**Theorem 7.2 (Construction of a Subprogram Body)**
*Let $T_{init} \subset \mathcal{T}_{\Sigma \cup \{\Omega\}}$ be a set of initial trees with $t_e \in T_{init}$ for all $e \in E$. Let $(U_{rec}, U_{sub})$ be a valid segmentation of $T_{init}$ with $U_{sub} = \emptyset$ and $G$ the function variable used for constructing the segments. The maximal pattern $\hat{t}_G \in \mathcal{T}_{\Sigma \cup G}(X)$ of all segments $\mathbf{segment}(t_e, U_{rec}, w)$ is the resulting hypothesis about the program body.*
*Proof: Follows from theorem 7.1 and definition 7.40.*

The variable instantiations for a maximal pattern $\hat{t}_G$ are given by the subtrees at the positions where the segmentations differ or (for uncomplete unfoldings) by $\perp$, if these positions do not occur in a given tree (see def. 7.40).

**Definition 7.42 (Variable Instantiations)** *Let $T \subset \mathcal{T}_{\Sigma \cup \{\Omega\}}$ be a set of terms indexed over $E$, $\hat{t}_G$ the maximal pattern of terms in $T$, and $X = \mathbf{var}(\hat{t}_G)$ the set of variables in the maximal pattern. The instantiation $\beta_e : X \to \mathcal{T}_\Sigma \cup \{\perp\}$ of variables $x \in X$ at positions $u \in \mathbf{pos}(\hat{t}_G, x)$ in term $t_e \in T$ is defined as*

$$\beta_e(x) = \begin{cases} t_e|_u & u \in \mathbf{pos}(t_e) \\ \perp & otherwise. \end{cases}$$

*Table 7.9.* Anti-Unification for Incomplete Segments

We define $t \sqcap \Omega = \Omega \sqcap t = t$.

**Complete Segment:** $t_1 = if(eq0(2), 1, if(eq0(pred(2)), 1, +(G, G)))$

**Incomplete Segment:** $t_2 = if(eq0(pred(pred(2))), 1, \Omega)$

**Anti-Unification:** $if(eq0(x_1), 1, if(eq0(pred(2)), 1, +(G, G)))$

### 3.2.3 ALGORITHM

The maximal pattern of a set of terms can be calculated by first order anti-unification as described in definition 7.16 in section 1.2. Only *complete* segments are considered. For incomplete segments, it is in general not possible to obtain a consistent introduction of variables during generalization. An example problem, using two terms of *Fibonacci* (introduced in table 7.3) is given in table 7.9.

Anti-unification gives us the maximal body $\hat{t}_G$ of a subprogram $G$. The variables in the subprogram represent that subtrees which differ over the segments of an initial tree. After this step of folding, the instantiations of these variables are still a *finite* set of terms, namely, the differing subtrees. In section 3.4 it will be shown, how this finite set is generalized by inferring input variables, their initial instantiation, and the substitution of these variables in the recursive call.

## 3.3 DEALING WITH FURTHER SUBPROGRAMS

Up to now we have ignored the case that there are path leading to $\Omega$s which cannot be explained by the constructed segmentation and that cannot be generated by the resulting maximal subprogram body. In this section we will present how folding works if $U_{sub}$ is not empty. First, we give an illustrative example. Then we introduce decomposition of initial trees with respect to positions in $U_{sub}$. We show that integrating calls to further subprograms in the body of a subprogram called from the main program $t_0$ is well-defined. Finally, we describe how the original initial tree can be reduced by replacing subtrees corresponding to calls of further subprograms by their name.

### 3.3.1 INDUCING TWO SUBPROGRAMS FOR 'MODLIST'

Remember the *ModList* example presented in table 7.2 and the example initial tree from which this RPS can be inferred given in figure 7.7. Let us look at this initial tree again (see fig. 7.8): We can find a first segmentation which explains the right-most path leading to an $\Omega$ with $U_{rec} = \{3.2.\lambda\}$. But, the initial tree contains further subtrees with $\Omega$-leafs, that is $U_{sub} = \{3.1.\lambda\} \neq \emptyset$. If the found first segmentation is valid, there must exist a

further subprogram which can generate these remaining subtrees containing $\Omega$s (marked with black boxes in the figure). In segments one to three (segment four is an incomplete unfolding, as discussed above), the left-hand subtrees of the 'cons'-nodes contain these yet unexplained subtrees.

Folding has started with inferring an RPS $\mathcal{S} = (\mathcal{G}, t_0)$ and a promising segmentation has been found which might result in a subprogram $G_1 \in \mathcal{G}$ and a main program $t_0$ which calls $G_1$. Now, the folding algorithm is called recursively with the unexplained subtrees as new sets of initial trees $T^u_{init}$ with $u \in U_{sub}$. For the *ModList* example, there exists only *one* position in $U_{sub}$ and we have three example trees which must be explained. The recursive call of the folding algorithm results again in finding a recursive program *scheme* $\mathcal{S}^u = (\mathcal{G}^u, t^u_0)$ and not just a recursive subprogram in $\mathcal{G}$. This is, because the call of the further sub-program might be embedded in a constant term $t^u_0$ which later becomes part of the body of the calling subprogram $G_1$. That is, we start to infer a *sub*-scheme which afterwards can be integrated in the searched-for RPS.

Again, the first step of folding is to find a valid recurrent segmentation. In figure 7.9 the new set of initial trees is given. The initial hypothesis, that $t^u_0$ consists only of the call of the subprogram fails. There is a constant initial part *if(eq0(x), T, nil)*. Starting segmentation further down, at the first 'if'-node, results in a valid recurrent segmentation which can explain all three initial trees.

We already described, how the maximal body can be constructed by finding the common pattern of all segments (see sect. 3.2. For the searched for subprogram $G_2$ we find the body *if(<(*$x_1$*, head(*$x_2$*)),* $x_1$*,* $G_2$*)*. We will explain, how initial instantiations and substitutions in recursive calls are calculated below, in section 3.4. For the moment, just believe that the resulting subprogram $G_2$ is the one given in the box in figure 7.9. The calling main program for the sub-scheme is $t^u_0 =$ *if(eq0(*$G_2$*(*$x_1$*,* $x_2$*)), T, nil)*. The initial trees $T^u_{init}$ can be explained with the following initial instantiations:

**First Tree:** $\beta(x_1) = 8$, $\beta(x_2) = [9, 4, 7]$,

**Second Tree:** $\beta(x_1) = 8$, $\beta(x_2) = tail([9, 4, 7])$,

**Third Tree:** $\beta(x_1) = 8$, $\beta(x_2) = tail(tail([9, 4, 7]))$.

Now the inference of the sub-scheme is complete and the folding algorithm comes back to the original problem. The originally unexplained subtrees are replaced by the three calls of the main program $t^u_0$ given at the bottom of figure 7.9. The inferred subprogram $G_2$ is introduced in the set of subprograms $\mathcal{G}$ of the to be constructed RPS $\mathcal{S}$. The *reduced* initial tree is given in figure 7.10. The folding algorithm proceeds now with constructing the program body for $G_1$.

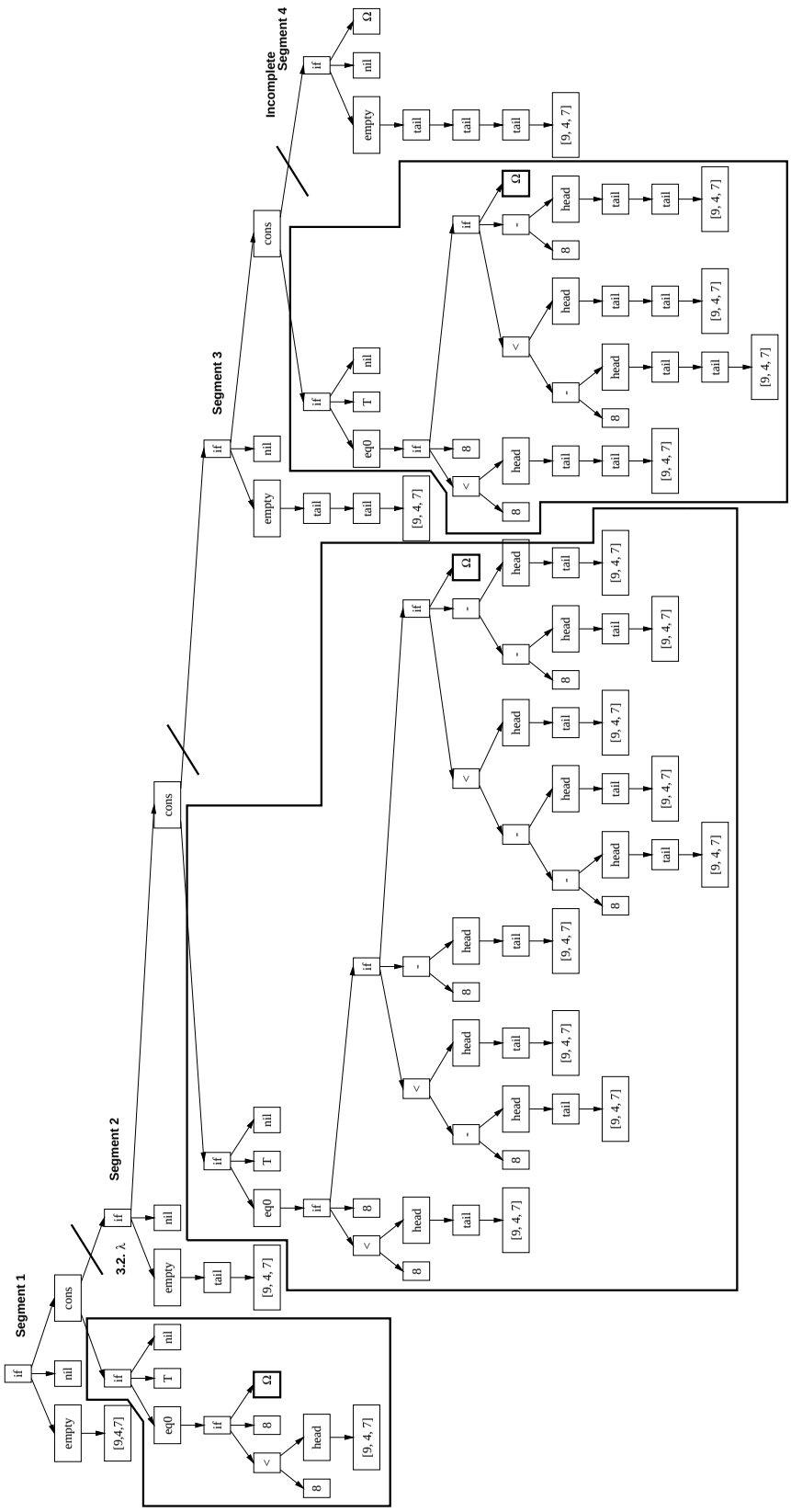



*Figure 7.8.* Identifying Two Recursive Subprograms in the Initial Program for *ModList*

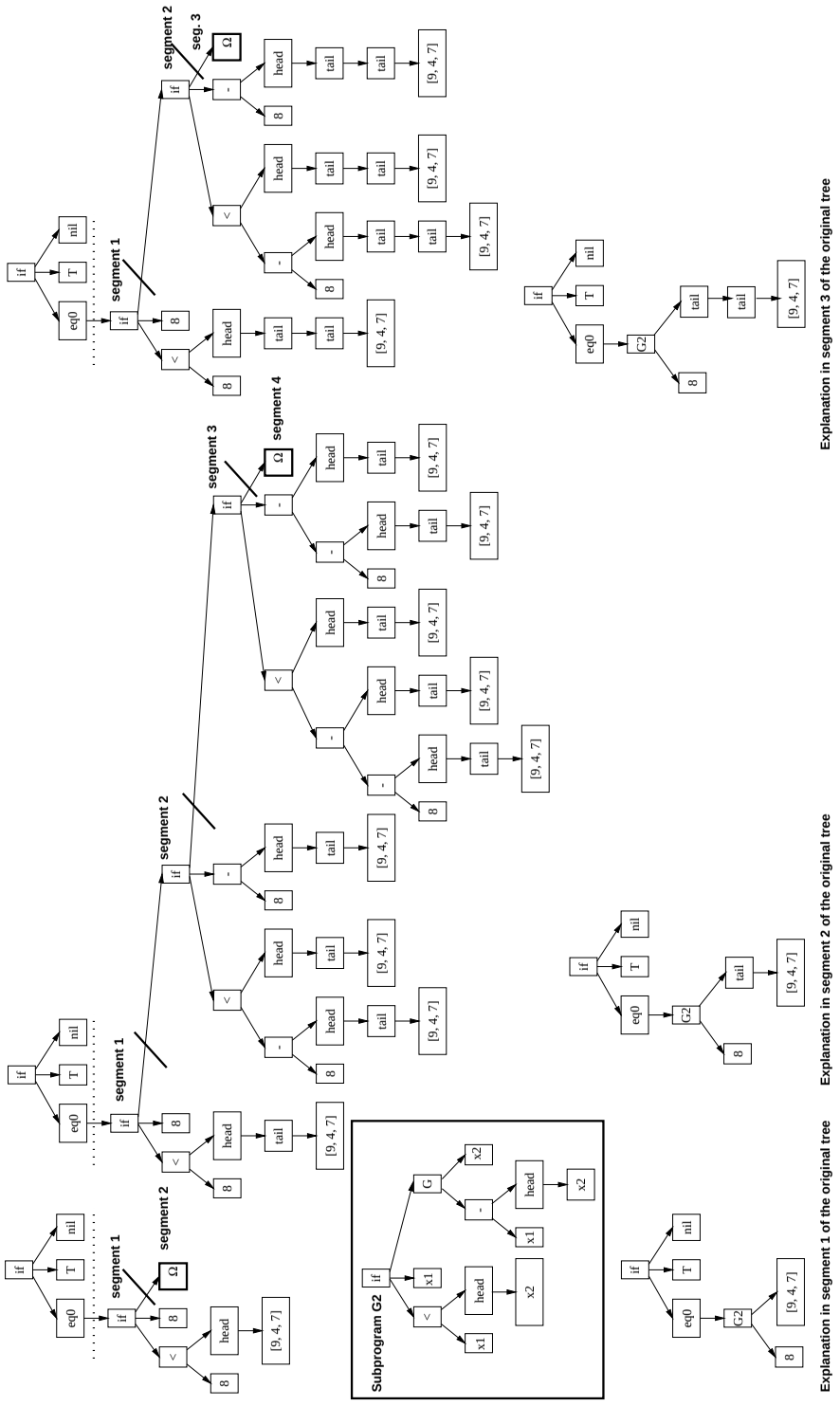



*Figure 7.9.* Inferring a Sub-Program Scheme for *ModList*

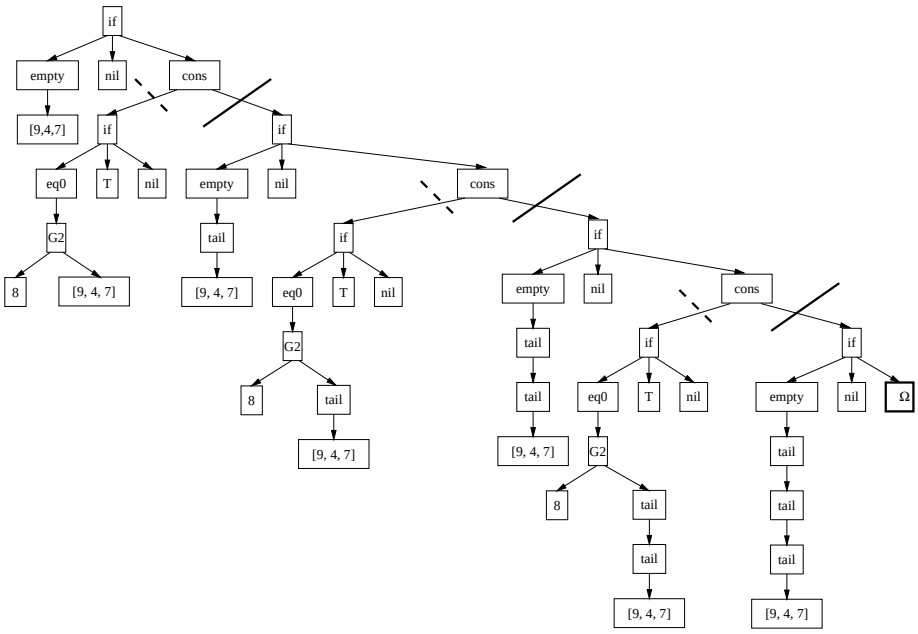



*Figure 7.10.* The Reduced Initial Tree of *ModList*

### 3.3.2 DECOMPOSITION OF INITIAL TREES

If a valid recurrent segmentation $(U_{rec}, U_{sub})$ was found for a set of initial trees $T_{init}$ and if $U_{sub} \neq \emptyset$, then induction of an RPS is performed recursively over the set of subtrees at the positions in $U_{sub}$.

**Definition 7.43 (Decomposition of Initial Trees)** *Let $T_{init} \subset \mathcal{T}_{\Sigma\cup\{\Omega\}}$ be a set of initial trees indexed over $E$. Let $(U_{rec}, U_{sub})$ be a valid segmentation of $T_{init}$ with $U_{sub} \neq \emptyset$. The set of initial trees $T^u_{init}$ for all $u \in U_{sub}$ is defined as:*

$$T^u_{init} = \{\mathbf{segment}(t_e, U_{rec}, w)|_u \mid \begin{array}{l} e \in E, w \in W(e), \\ u \in \mathbf{pos}(\mathbf{segment}(t_e, U_{rec}, w))\}. \end{array}$$

*The index set $E^u$ of elements in $T^u_{init}$ is constructed as $E^u = \{(w,e) \mid e \in E, w \in W(e), u \in \mathbf{pos}(\mathbf{segment}(t_e, U_{rec}, w))\}$ and for elements $t_{(w,e)} \in T^u_{init}$ with $(w,e) \in E^u$ holds $t_{(w,e)} = \mathbf{segment}(t_e, U_{rec}, w)|_u$.*

From the construction of $(U_{rec}, U_{sub})$ follows that each subtree at a position in $U_{sub}$ contains at least one $\Omega$ (see def. 7.36 and def. 7.37). An example for an initial tree with non-empty sub-schema positions was given in figure 7.7.

### 3.3.3 EQUIVALENCE OF SUB-SCHEMATA

In general, there can exist alternative RPSs which can explain a set of initial trees. These RPSs might have different numbers of parameters in the main program with different initial instantiations. We want to deal with induction of sub-schemata for subtrees of a set of given initial trees as independent sub-problem. For that reason, it must hold that – if there exists a valid hypothesis for initial trees in $T^u_{init}$ which is part of a recursive explanation of $T_{init}$ – *each possible solution* for $T^u_{init}$ can be part of the recursive explanation of $T_{init}$.

For understanding the following theorem, remember that the main program $t^u_0$ of a sub-scheme must be valid over different examples for this scheme. For the *ModList* illustration (see fig. 7.9) there were three given trees for inferring the sub-scheme and a main program $t^u_0$ could be generated covering all three trees with different initial instantiations $\beta(t^u_0)$.

**Theorem 7.3 (Equivalence of Sub-Schemata)**
*Let $T_{init}$ be a set of initial trees indexed over $E$. Let $\mathcal{S}^1 = (t^1_0, \mathcal{G}^1)$ and $\mathcal{S}^2 = (t^2_0, \mathcal{G}^2)$ be two RPSs with subprograms $G^1_1, \ldots, G^1_{n_1}$ and $G^2_1, \ldots, G^2_{n_2}$. The RPSs together with initial instantiations $\beta^1_e : X^1 \to \mathcal{T}_\Sigma$ and $\beta^2_e : X^2 \to \mathcal{T}_\Sigma$ for all $e \in E$ and parameters $X^1 = \mathbf{var}(t^1_0)$ and $X^2 = \mathbf{var}(t^2_0)$ recursively explain $T_{init}$. Let $t^1_0$ be the maximal pattern of all terms $\beta^1_e(t^1_0)$ and $t^2_0$ the maximal pattern of all terms $\beta^1_e(t^2_0)$ for $e \in E$. It holds that for each parameter $x^1 \in X^1$ exists a parameter $x^2 \in X^2$ such that for all $e \in E$: $\beta^1_e(x^1) = \beta^2_e(x^2)$.*

**Proof 7.1 (Equivalence of Sub-Schemata)**
*Let $\mathcal{S}^1$ be a sub-schema constructed by the "maximization of body" principle (see theorem 7.1) and $\mathcal{S}^2$ a sub-schema which is obtained in some different way, that is, recursion points must be at "deeper" positions in the initial trees.*
*Let $x^1 \in X^1$ be a parameter of $t^1_0$. For instantiations $\beta^1_e(x^1)$ holds that they do not share a common non-trivial pattern over all positions of $x^1$ in the segments of the initial trees. It holds $\exists e, e' \in E$ with $\mathbf{node}(\beta^1_e(x^1)) \neq \mathbf{node}(\beta^1_{e'}(x^1))$ (postulation in theorem 7.3).*
*Variable $x^1$ is used in RPS $\mathcal{S}^1$ to generate sub-terms of the given initial trees. Let $u_{x^1}$ be a position in initial trees $t_e$, $e \in E$, where terms are generated by instantiations $\beta^1_e(x^1)$. The sub-terms at positions $t_e|_{u_{x^1}}$ are identical to the initial instantiation $\beta^1_e(x^1)$ and do not share a non-trivial pattern. It follows, that in RPS $\mathcal{S}^2$, the sub-terms $t_e|_{u_{x^1}}$ must also be explained by an initial instantiation in main ($t^2_0$) (*case 1*) or be part of an instantiated parameter occuring in an unfolding of a subprogram (*case 2*).*

**Case 1:** *(sub-terms $t_e|_{u_{x^1}}$ are explained by an initial instantiation in $t^2_0$)*
*Let $x^2 \in X^2$ be a variable and $u_{x^2} \in \mathbf{pos}(t^2_0\{G^2_1 \leftarrow \Omega, \ldots, G_{n_2} \leftarrow \Omega\}, x^2)$ be a position of this variable in main with $u_{x^2} \leq u_{x^1}$. Because $x^2$ was introduced by anti-unification, exist $e, e' \in E$ with $\mathbf{node}(u_{x^2}, t_e) \neq \mathbf{node}(u_{x^2}, t_{e'})$. Because for all $u < u_{x^1}$ holds that $\mathbf{node}(u, t_e) = \mathbf{node}(u, t_{e'})$ for all $e, e' \in E$ follows $u_{x^2} = u_{x^1}$ and therefore $\beta^1_e(x^1) = \beta^2_e(x^2)$.*

**Case 2:** *(sub-terms $t_e|_{u_{x^1}}$ are part of a parameter instantiation in an unfolding of a subprogram from $\mathcal{S}^2$)*

*Let $u_S$ be a position with $u_S \leq u_{x^1}$ such that terms $t_e|_{u_{x^1}}$ are instantiations of a parameter in an unfolding of a subprogram from $\mathcal{S}^2$. These instantiations are generated by a sequence of substitutions (in the call of a subprogram from main, in the recursive call of a subprogram, in the call of a further subprogram, ...). Let $t_{subst} \in \mathcal{T}_\Sigma(X^2)$ be a combination of such substitutions where variables in $X^2$ are parameters of $t_0^2$. That is, it holds $t_e|_{u_S} = t_{subst}[\beta_e^2(x_1^2), \ldots, \beta_e^2(x_m^2)] = \beta_e^2(t_{subst})$ for all $e \in E$.*
*Let $u^s_{x^1}$ be the position of sub-term $t_e|_{u_{x^1}}$ with respect to position $u_S$, that is $(t_e|_{u_S})|_{u^s_{x^1}} = t_e|_{u_{x^1}}$. Because sub-terms $t_e|_{u_{x^1}}$ for all $e \in E$ and therefore sub-terms $\beta_e^2(t_{subst})|_{u^s_{x^1}}$ do not share a non-trivial pattern, terms $\beta_e^2(t_{subst})|_{u^s_{x^1}}$ must be part of instantiations of variables $x^2 \in X^2$. Let $u^s_{x^2}$ be a position of variables $x^2$ in $t_{subst}$ with $u^s_{x^2} \leq u^s_{x^1}$ and $t_{subst}|_{u^s_{x^2}} = x^2$. Because for each position $u < u^s_{x^1}$ holds $\mathbf{node}(\beta_e^2(t_{subst})|_u) = \mathbf{node}(\beta_{e'}^2(t_{subst})|_u)$ for all $e, e' \in E$ and because $\beta_e(x^2)$ do not share a common prefix (postulation) must hold $\beta_e^2(x^2) = \beta_2^2(t_{subst})|_{u^s_{x^1}} = t_e|_{u_{x^1}} = \alpha_e^1(x^1)$.*

Theorem 7.3 states that for each RPSs with a given initial instantiation of variables in the main program which explains the set of subtrees at positions $U_{sub}$ results an identical instantiation of variables. Therefore, independent of the induced subprogram together with a calling main program, the number and instantiations of parameters occuring in this main program are unique. The proof is based on the idea that for a given set of initial trees and a given sub-schema $\mathcal{S}^1$ each other sub-schema $\mathcal{S}^2$ which explains the same initial trees shares certain characteristics from which follows that parameter instantiations are identical. In the first case, we consider parameter instantiations which are part of the instantiation of parameters in the calling main program. If $u_{x^2}$ is above $u_{x^1}$ in the initial trees then the subtrees must differ at position $u_{x^2}$. But we already know that $u_{x^1}$ is exactly that position at which the subtrees differ the first time. Therefore $u_{x^1}$ and $u_{x^2}$ must be identical positions in the subtrees and as a consequence the instantiations which are given as the subtrees at these positions must be identical! Case two is analogous for sub-terms in segments corresponding to unfoldings of a subprogram in $\mathcal{S}^2$.

For illustration consider again the *ModList* example given in figure 7.10 and the searched-for RPS given in table 7.2. The subprogram *ModList* with parameters $l$ and $n$ calls a further subprogram *Mod*. The "partial" program body of *ModList*, that is, the maximal pattern of all segments without consideration of further subprograms, is:

$$ModList = if(empty(l), nil, cons(u, ModList)).$$

For the set of subtrees at segment positions $u = 3.1.\lambda$ for example the *Mod* subprogram given in table 7.2 can be induced with parameters $k$ and $n$. For this subprogram the calling main program is *if(eq0(Mod(n, head(l))), T, nil)* with parameter $n$ being passed through from the main program of the RPS and parameter $k$ being constructed by substituting the parameter $l$ given in the main program by *head(l)*. Because *ModList* as well as *Mod* is constructed by

calculating the maximal pattern over all segments, each possible realization of *Mod* together with its calling main *must* contain parameters $n$ and *k = head(l)*.

### 3.3.4 REDUCTION OF INITIAL TREES

If sub-schemata explaining all subtrees $T^u_{init}$ can be induced for all positions $U_{sub}$, these sub-schemata can be inserted in the segments of the subprogram which calls the sub-schema. That is, the concerned subtrees are replaced by the main programs $t^u_0$ which call further sub-programs.

**Lemma 7.2 (Reduction of Initial Trees)**
*Let $T_{init}$ be a set of initial trees indexed over $E$. Let $(U_{rec}, U_{sub})$ together with $t_{skel}$ be a valid segmentation for all $t_e \in T_{init}$ and let $U_{sub} \neq \emptyset$.*

1. *If there exists an RPS $\mathcal{S}^u = (\mathcal{G}^u, t^u_0)$ for each $u \in U_{sub}$ which explains the initial trees $T^u_{init}$ constructed as described in definition 7.43 together with initial instantiations $\beta^u_{(w,e)} : \mathbf{var}(t^u_0) \to \mathcal{T}_\Sigma \cup \{\bot\}$[8], the segments of initial trees in $T_{init}$ can be transformed by $\mathbf{segment}(t_e, U_{rec}, w) = \mathbf{segment}(t_e, U_{rec}, w)[u \leftarrow \beta^u_{(w,e)}(t^u_0)]$ for all trees $e \in E$, unfolding indices $w \in W(e)$, and sub-schema positions $u \in \mathbf{pos}(\mathbf{segment}(t_e, U_{rec}, w))$.*
2. *Let be $\mathcal{G}^u$ a set of properly named subprograms (no two subprograms have identical names $G_i$) with*

$$\mathcal{G}^u = \begin{array}{ll} \langle G^u_1(x_1, \ldots, x_{m^u_1}) & = t^u_1, \\ \vdots & \\ G^u_n(x_1, \ldots, x_{m^u_n}) & = t^u_n \rangle \end{array}$$

*and let $G(x_1, \ldots, x_m) = t_G$ be the (yet unknown) subprogram for segmentation $(U_{rec}, U_{sub})$. The RPS $\mathcal{S} = (\mathcal{G}, t_0)$ which explains $T_{init}$, contains all subprograms of the sub-schemata $\mathcal{S}^u = (\mathcal{G}^u, t^u_0)$ for all $u \in U_{sub}$:*

$$\mathcal{G} = \begin{array}{ll} \langle G(x_1, \ldots, x_m) & = t_G, \\ G^u_1(x_1, \ldots, x_{m^u_1}) & = t^u_1, \\ \vdots & \\ G^u_n(x_1, \ldots, x_{m^u_n}) & = t^u_n \rangle. \end{array}$$

**Proof 7.2 (Reduction of Initial Trees)**

1. *If an RPS $\mathcal{S}^u = (\mathcal{G}^u, t^u_0)$ together with an initial instantiation $\beta^u_{(w,e)}$ explains an initial tree $t^u_{(w,e)}$, then there exists a term $t \in \mathcal{L}(\mathcal{S}^u, \beta^u_{(w,e)})$ with $t^u_{(w,e)} \leq_\Omega t$. Therefore, if the subtree at position $u$ in segment $\mathbf{segment}(t_e, U_{rec}, w)$ is replaced by the instantiated calling main*

[8] There can exist subtrees for the sub-schemata which are too small to infer all variables.

*program* $\beta^u_{(w,e)}(t^u_0)$*, then a term* $t'$ *with* $\mathbf{segment}(t_e, U_{rec}, w) \leq_\Omega t'$ *can be generated from* $\mathbf{segment}(t_e, U_{rec}, w)[u \leftarrow \beta^u_{(w,e)}(t^u_0)]$ *using rules of the term rewrite system* $\mathcal{R}_{\mathcal{S}^u}$.

2. *To generate the original segments (or larger subtrees) from the segments where at all positions* $u \in U_{sub}$ *the subtrees are replaced by the instantiated calling main programs, all subprogram definitions of the sub-schemata* $\mathcal{S}^u = (\mathcal{G}^u, t^u_0)$ *are necessary.*

After reduction, in the segments of $T_{init}$ the subtrees at positions $U_{sub}$ are replaced by the instantiated calling main of the corresponding subprograms. That is, these subtrees do no longer contain $\Omega$s and the program body of the segments can be constructed as described in section 3.2.

The prescribed construction of sub-programs imposes the following restriction on initial trees: A maximal pattern for the subprograms can only be constructed if there exist at least two trees $t_{(w,e)}, t_{(w',e')} \in T^u_{init}$ such that for the initial instantiation of all variables $x \in \mathbf{var}(t^u_0)$ holds $\beta^u_{(e,w)}(x) \neq \perp$ and $\beta^u_{(e',w')}(x) \neq \perp$.

A consequence of inferring sub-schemata separately is, that subprograms are introduced *locally* with unique names. It is possible, that two such subprograms are identical and have only different names. After folding is completed the set of subprograms can be reduced such that only subprograms with different program bodies remain in $\mathcal{G}$.

## 3.4 FINDING PARAMETER SUBSTITUTIONS

After constructing a subprogram body as maximal pattern of all segments, the remaining not explained subtrees must be parameters and their substitutions. Anti-unification already results in a set of variables. The last component of inducing an RPS from a set of initial trees is to identify the substitutions these variables in the recursive calls of the subprogram.

Variable substitutions in recursive calls can be quite complex. In table 7.10 some recursive equations with different variants of substitutions are given. In the most simple case, each variable is substituted independently of the other variables and keeps its position in the recursive call ($f_1$). A variable might be also substituted by an operation involving other program parameter ($f_2$). Additionally, variables can switch there positions (given in the head of the equation) in the recursive call ($f_3$). Finally, there might be "hidden" variables which only occur within the recursive call ($f_4$). If you look at the body of $f_4$, variable $z$ occurs only within the recursive call. The existence of such a variable cannot be detected when the program body is constructed by anti-unification but only a step later, when substitutions for the recursive call are inferred.

### 3.4.1 FINDING SUBSTITUTIONS FOR 'MOD'

Let us again look at the *Mod* example. The valid segmentation for the second initial trees given in figure 7.9 is depicted again in figure 7.11. We already know

*Table 7.10.* Variants of Substitutions in Recursive Calls

*Individual Substitutions, Identical Positions*
f1(x, y) = if(eq0(x), y, +(x, f1(pred(x),succ(y))))

*Interdependent Substitutions*
f2(x, y) = if(eq0(x), y, +(x, f2(pred(x),+(x,y))))

*Switching of Variables*
f3(x, y, z) = if(eq0(x), +(y, z), +(x, f3(pred(x), z, succ(y))))

*Hidden Variable*
f4(x, y, z) = if(eq0(x), y, +(x, f4(pred(x), z, succ(y))))

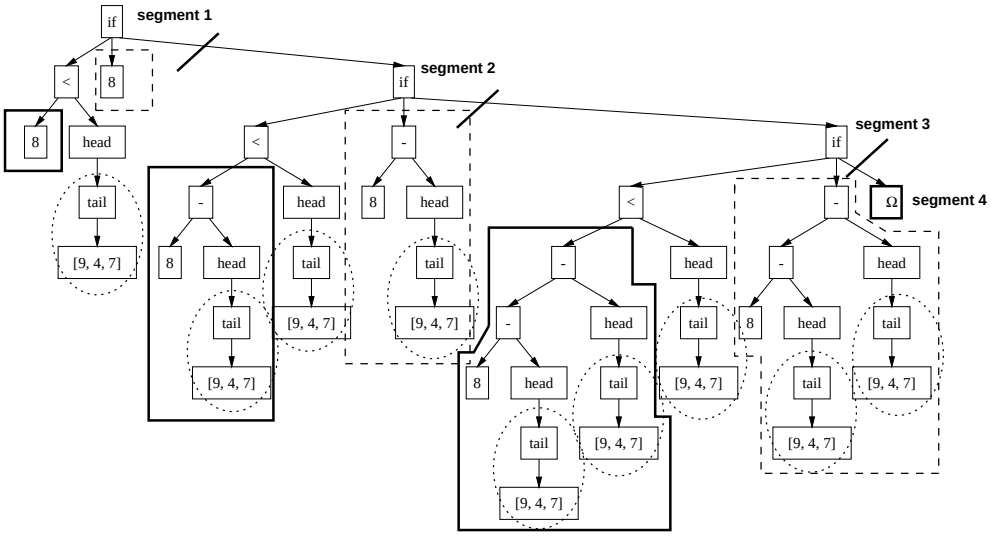



*Figure 7.11.* Substitutions for *Mod*

from calculating the subprogram body *if(<(*$x_1$*, head(*$x_2$*))*, $x_1$, $G_2$*)* that there remain two different kinds subtrees which must be explained by variables and their substitutions in recursive calls. Remember, that in constructing the subprogram body for *Mod*, there were three initial trees at hand and the program body was constructed by anti-unifying the segments gained from all three initial trees. Variable $x_1$ reflects differences appearing already in segments of a single of such initial trees, but variable $x_2$ reflects differences between segments of different initial trees.

For better readability we write the remaining subtrees for each segment of the given tree into a table:

| $x_1$ (1.1.1.$\lambda$) | $x_2$ (1.1.2.1.$\lambda$) | $x_3 = x_1$ (1.2.$\lambda$) |
|---|---|---|
| 8 | tail([9,4,7]) | 8 |
| −(8, head(tail([9,4,7]))) | tail([9,4,7]) | −(8, head(tail([9,4,7]))) |
| −(−(8, head(tail([9,4,7]))), head(tail([9,4,7]))) | tail([9,4,7]) | −(−(8, head(tail([9,4,7]))), head(tail([9,4,7]))) |

The middle column represents the instantiations of $x_2$ which is constant over all segments. If only this single initial tree were be considered, this term would be part of the program body. The left-hand and right-hand columns contain exactly the same terms on each level. With these considerations, it is enough to consider the changes from one level to the next in the first two columns. Terms on succeeding levels represent changes from one segment to the next, that is, for the program body induced by this segmentation, these changes must be explained by substitutions of parameters in the recursive call.

When going from the first to the second level, we find an occurrence of $x_1$ as well as of $x_2$, that is *−(8, head(tail([9,4,7]))) = −(*$x_1$*, head(*$x_2$*))*. This leads to the hypothesis, that the initial instantiations are $x_1 = 8$ and $x_2 = tail([9,4,7])$ with substitutions $x_1 \leftarrow$ *−(*$x_1$*, head(*$x_2$*))* and $x_2 \leftarrow x_2$. We used the first two segments to *construct* a hypothesis. As we will see below, in general, we need two further segments to *validate* this hypothesis. For now, let it be enough that we validate the hypothesis by going from the second to the third segment. Applying the found substitution to $x_1 =$ *−(8, head(tail([9,4,7])))* results in *−(−***(8, head(tail([9,4,7])))**, *head(tail([9,4,7])))* and because $x_2 =$ *tail([9,4,7])*, the substitution hypothesis holds!

The illustration gives an example for an interdependent substitution (see tab. 7.10) because the substitution of $x_1$ depends not only on $x_1$ itself, but also on $x_2$.

### 3.4.2 BASIC CONSIDERATIONS

A necessary characteristic of substitutions is that they are unambiguously determined in the (infinite set of) unfoldings (see def. 7.29). In that case, substitutions form regular patterns in the initial trees and therefore can be identified by pattern matching.

**Theorem 7.4 (Uniqueness of Substitutions)**
*Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS, $G_i(x_1, \ldots, x_{m_i}) = t_i$ a subprogram in $\mathcal{S}$, and $X_i = \{x_1, \ldots, x_{m_i}\}$ the set of parameters. Let $U_{rec}$ be the set of recursion points of $G_i$ with index set $R = \{1, \ldots, |U_{rec}|\}$, $W$ the unfolding indices constructed over $R$, and $\Upsilon_i$ the set of all unfoldings of equation $G_i$ over instantiations $\beta : X_i \to \mathcal{T}_\Sigma$. Let $t_i$ be a program body which corresponds to the maximal pattern of all unfoldings $\upsilon_w \in \Upsilon_i$. Let $\mathbf{sub}(x_j, r)$ be a substitution term (def. 7.27) for $x_j \in X_i$ in the recursive call of $G_i$ at position $u_r \in U_{rec}$, $r \in R$. For all terms $t_s \in \mathcal{T}_\Sigma(X)$ with $\forall w \in W : \beta_{w \circ r}(x_j) = t_s\{x_1 \leftarrow$*

$\beta_w(x_1), \ldots, x_{m_i} \leftarrow \beta_w(x_{m_i})\}$ *holds* $t_s = \mathbf{sub}(x_j, r)$.
*Proof: see appendix B3.*

Because substitutions are unique over unfoldings, it holds that for each subtree under a recursion point it is determined whether this term is a parameter with its initial instantiation or a substitution term over such a parameter.

**Definition 7.44 (Characteristics of Substitution Terms)** *Let* $\mathbf{sub}(x_j, r)$ *be a substitution term for parameter* $x_j$ *at position* $r$. *For all positions* $u$, *starting with* $u = \lambda$, *holds the following inductively defined characteristic:*

$$\mathbf{sub}(x_j, r)|_u = \begin{cases} x_k & (a)\ \forall w \in W : \beta_{(w \circ r)}(x_j)|_u = \beta_w(x_k)\ and \\ & (b)\ \neg\exists t \in \mathcal{T}_\Sigma(X_i)\ with\ t \neq x_k,\ such\ that\ holds \\ & \forall w \in W : \beta_{w \circ r}(x_j)|_u = \\ & t\{x_1 \leftarrow \beta_w(x_1), \ldots, x_{m_i} \leftarrow \beta_w(x_{m_i})\} \\ f(\mathbf{sub}(x_j, r)|_{u \circ 1}, \ldots, & \forall w \in W : \mathbf{node}(\beta_{w \circ r}(x_j), u) = f \in \Sigma \\ \mathbf{sub}(x_j, r)|_{u \circ n}) & with\ arity\ \alpha(f) = n. \end{cases}$$

For sufficiently large initial trees, it can be decided for each parameter in the $r$-th unfolding how it is substituted in the next $r \circ w$-th unfolding. For example, in a recursive equation with two parameters $x_1$ and $x_2$ with $\beta(x_1) = 0$ and $\beta(x_2) = succ(0)$, both parameters might be substituted by $x_i \leftarrow succ(x_i)$. Condition *(a)* alone allows that $x_1 \leftarrow x_2$ is identified as substitution, but this is only a legal substitution, if condition *(b)* holds additionally. For the given example, for $x_1$ in the $r$-th unfolding can be found a substitution term $succ(x_1)$ in the $r \circ w$-th unfolding. A function for testing uniqueness of substitution is given in table 7.11. The function makes use of a function which tests the recurrence of a substitution given in table 7.12.

For a given hypothesis $\hat{t}_G$ about a subprogram body (see def. 7.2 in sect. 3.2), variable instantiations can be determined for each segment. After determining variable instantiations for segments, the initial instantiations can be separated from the substitution terms by calculating the differences between the segment-wise instantiations. To calculate variable instantiations for segments, we can extend the definition for parameter instantiations (def. 7.18) in the following way:

**Definition 7.45 (Instantiation of Variables in a Segment)** *Let* $T_{init} \subset \mathcal{T}_{\Sigma \cup \{\Omega\}}$ *be a set of initial trees indexed over* $E$. *Let* $(U_{rec}, U_{sub})$ *with* $U_{sub} = \emptyset$ *be a segmentation of trees in* $T_{init}$ *and* $\hat{t}_G$ *the resulting maximal pattern with* $X = \mathbf{var}(\hat{t}_G)$. *Let* $W(e)$ *be the set of segment indices for* $t_e \in T_{init}$. *The instantiations* $\beta : X \times W \times E \rightarrow \mathcal{T}_\Sigma \cup \{\perp\}$ *for variables in*

*Table 7.11.* Testing whether Substitutions are Uniquely Determined

**Function:** $\mathbf{uniqueness} : X \times X \times \mathbf{pos}(t_{init}) \times \mathcal{T}_\Sigma \to \mathbf{bool}$

**Pre:** $E$ is the set of indices for $T_{init}$
$R$ is the index set of recursion points
$W(e)$ ist the set of segment indices for $t_e \in T_{init}$
$X$ is the set of variables in $\hat{t}_G$ and $x_j, x_k \in X$
$\beta(x_k, w, e)$ returns the instantiation of variable $x_k \in X$ for segment $w \in W(e)$
$t_\beta$ is an instantiation of $x_j \in X$ with $t_\beta \neq \bot$ in a segment with $w = r.\lambda$
$u$ is the position which is currently checked

**Post:** $\mathbf{true}$ if for $x_j$ cannot be found a recurrent substitution for a variable from $X \setminus \{x_k\}$, $\mathbf{false}$ otherwise

**Side-effect:** Break if an ambiguity is detected

$\mathbf{uniqueness}(x_j, x_k, u, t_\beta)$ =

1. **IF** $u \notin \mathbf{pos}(t_\alpha)$
2. **OR** $((\neg\exists x_i \in (X \setminus \{x_k\}) : \mathbf{betaRecurrent}(x_j, x_i, u))$ (see table 7.12)
3. **OR** $(\neg\forall e, e' \in E, w \in W(e), w' \in W(e') : \mathbf{node}(\beta(x_j, w \circ r, e), u) = \mathbf{node}(\beta(x_j, w' \circ r, e'), u)))$
4. **AND Let** $f = \mathbf{node}(t_\beta, u)$ with $\alpha(f) = n$ **In**
   $\forall l = 1, \ldots n : \mathbf{uniqueness}(x_j, x_k, u \circ l, w_1, e_1)$
5. **THEN return TRUE**
6. **ELSE return FALSE** and **break**

*Table 7.12.* Testing whether a Substitution is Recurrent

**Function:** $\mathbf{betaRecurrent} : X \times X \times R \times \mathbf{pos}(t_{init}) \to \mathbf{bool}$

**Pre:** $E$ is the set of indices for $T_{init}$
$R$ is the index set of recursion points
$W(e)$ ist the set of segment indices for $t_e \in T_{init}$ and $w \in W(e)$
$X$ is the set of variables in $\hat{t}_G$ and $x_j, x_k \in X$
$\beta(x_k, w, e)$ returns the instantiation of variable $x_k \in X$ for segment $w \in W(e)$
$t_\beta$ is an instantiation of $x_j \in X$ with $t_\beta \neq \bot$ in a segment with $w = r.\lambda$
$u$ is the position which is currently checked

**Post:** $\mathbf{true}$ if in all instantiations of $x_j \in X$ in segment $w \circ r$ at position $u$ it can be found the instantiation of $x_k$ in segment $w$, $\mathbf{false}$ otherwise

$\mathbf{betaRecurrent}(x_j, x_k, r, u)$ =

$\forall e \in E, \forall w \circ r \in W(e)$ :
$\beta(x_j, w \circ r, e)|_u =_\bot \beta(x_k, w, e)$
(see def. 7.45 for $=_\bot$)

*$X$ occuring in segments with indices $w \in W(e)$ are defined as*

$$\beta(x,w,e) = \begin{cases} \mathbf{segment}(t_e, U_{rec}, w)|_u & (\mathbf{a})\ \exists u \in \mathbf{pos}(\hat{t}_G, x)\ with \\ & \quad u \in \mathbf{pos}(\mathbf{segment}(t_e, U_{rec}, w)) \\ & and \\ & (\mathbf{b})\ \mathbf{segment}(t_e, U_{rec}, w) \neq \bot \\ \bot & otherwise. \end{cases}$$

*With $\beta(x,w,e) =_\bot \beta(x',w',e')$ we denote that two instantiations are either identical or that one of the instantiations is undefined.*

Because we allow for incomplete unfoldings, it can happen that in some segments the positions of a hypothetical variable does not exist. Instantiation $\beta$ is $\bot$ (i. e., undefined), if the searched for position is non-existent in *all* segments.

Inducing the substitutions is based on the same principle as inducing the subprogram body, that is, to detect regularities in the yet unexplained subtrees of the segments. We realize the construction of a hypothesis for variable substitutions in the recursive calls of a subprogram by identifying instantiation and substitution through comparison of two successive segments ($w$ and $w \circ r$) and validate this hypothesis for all other pairs of successive segments ($w'$ and $w' \circ r$). For validation there must exist at least two additional successive segments where the position of the substitution term is given. Therefore, it must hold that the initial trees in $T_{init}$ must provide at least four segments for constructing and validating the searched for substitution of a variable. But it is not necessary for these further segments to be given in the *same* initial tree, it is enough that this information can be collected over all trees in $T_{init}$! Thereby we keep the restrictions on initial trees minimal. Furthermore, for calculating interdependent substitutions (see table 7.10), it is necessary, that for each substitution of a variable $x_j$ in a given segment, the substitution terms of *all* variables are defined in the preceding segment.

**Lemma 7.3 (Necessary Condition for Substitutions)**
*Let $T_{init} \subset \mathcal{T}_{\Sigma \cup \{\Omega\}}$ be a set of initial trees indexed over $E$ and $\hat{t}_G$ the hypothesis of the program body which follows from segmentation $(U_{rec}, U_{sub})$ with $U_{sub} = \emptyset$. Recursion points are indexed over $R = \{1, \ldots, |U_{rec}|\}$. The variables occuring in the program body are $X = \mathbf{var}(\hat{t}_G)$. Recurrent hypotheses over substitutions can only be induced if for each variable $x_j \in X$ and each recursive call $r \in R$ holds: $\exists e_1, e_2 \in E$ and $w_1 \in W(e_1)$, $w_2 \in W(e_2)$ with $w_1 = \lambda$, $(w_1, e_1) \neq (w_2, e_2)$ and for two positions $h \in \{1,2\}$:*

1. *$\beta(x_j, w_h \circ r, e_h) \neq \bot$ and*
2. *$\forall x_k \in X : \beta(x_k, w_h \circ r, e_h) \neq \bot$.*

*Table 7.13.* Testing the Existence of Sufficiently Many Instances for a Variable

**Function:** $\mathbf{enoughInstances} : X \to \mathbf{bool}$

**Pre:** $E$ is the set of indices for $T_{init}$
$R$ is the index set of recursion points
$W(e)$ ist the set of segment indices for $t_e \in T_{init}$
$X$ is the set of variables in $\hat{t}_G$ and $x_j \in X$
$\beta(x_k, w, e)$ returns the instantiation of variable $x_k \in X$ for segment $w \in W(e)$

**Post:** **true** if there are sufficiently many instances, **false** otherwise

$\mathbf{enoughInstances}(x_j)$ =

1. $(\exists e_1 \in E : \beta(x_j, r.\lambda, e_1) \neq \bot$ **AND**
2. $\quad \forall x_k \in X : \beta(x_k, \lambda, e_1) \neq \bot)$ **AND**
3. $(\exists e_2 \in E : \exists w \in W(e_2) : \beta(x_j, w \circ r, e_2) \neq \bot$ **AND**
4. $\quad \forall x_k \in X : \beta(x_k, w, e_2) \neq \bot)$ **AND**
5. $(e_1 \neq e_2$ **OR** $w \neq \lambda)$

An algorithm for testing whether enough instances are given in a set of initial trees is given in table 7.13.

After construction of a substitution hypothesis from a pair of succeeding substitutions and validating it for all (at least one further) pairs of substitutions for which the according sub-term positions are defined in the initial trees, it must be checked whether the complete set of substitution terms is consistent.

**Lemma 7.4 (Consistency of Substitutions)**
*Let $T_{init}$ be a set of initial trees indexed over $E$. Let be $\beta : X \times W \times E \to \mathcal{T}_{\Sigma \cup \{\bot\}}$ the set of instantiations of variables $x \in X$ in segments $e \in W(e), e \in E$. Let be $\mathbf{sub}(x_j, r)$ the substitution terms of variables $x_j \in X$ in the recursive calls. The composition of substitutions $\mathbf{sub}^* : X \times W \to \mathcal{T}_\Sigma(X)$ is inductively defined as:*

$$\mathbf{sub}^*(x_j, w) = \begin{cases} x_j & w = \lambda \\ \mathbf{sub}(x_j, r)\{x_1 \leftarrow \mathbf{sub}^*(x_1, v), \ldots, & \\ x_m \leftarrow \mathbf{sub}^*(x_m, v)\} & w = v \circ r. \end{cases}$$

*For each initial tree $t_e \in T_{init}$ must exist an instantiation $\beta(x_j, \lambda, e)$ such that for all variables $x_j \in X$ and all segments $w \in W(e)$ holds: $\beta(x_j, \lambda, e) =_\bot \mathbf{sub}^*(x_j, w)\{x_1 \leftarrow \beta(x_1, \lambda, e), \ldots, x_m \leftarrow \beta(x_m, \lambda, e)\}$.*

A substitution of a variable $x_j$ is consistent, if the substitution for a given segment at position $w$ can be constructed by successive application of the hypothetical substitution on the initial instantiation of the variable in the root segment ($w = \lambda$).

### 3.4.3 DETECTING HIDDEN VARIABLES

Remember that the set of parameters of a recursive subprogram initially results from constructing the hypothesis of the program body by anti-unification of the hypothetical segments (see sect. 3.2). The maximal pattern of all segments contains variables at all positions where the subtrees differ over the segments. In many cases, the set of variables identified in this way is already the final set of parameters of the to be constructed recursive functions. Only in the case of hidden variables (see table 7.10), that is, variables which only occur in substitution terms, the set of variables must be extended.

In contrast to the standard case, the instantiation of such hidden variables cannot be identified in the segments, they must be identified within the substitution terms instead. This can be done because, if such a variable occurs in a recursive equation it must be used in parameter substitutions in the recursive call.

**Lemma 7.5 (Identification of Hidden Variables)**
*Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS, $G_i(x_1, \ldots, x_{m_i}) = t_i$ a subprogram in $\mathcal{S}$, and $X_i = \{x_1, \ldots, x_{m_i}\}$ the set of parameters of $G_i$. Let $U_{rec}$ be the set of recursion points of $G_i$ indexed over $R = \{1, \ldots, |U_{rec}|\}$, $W$ the set of unfolding indices constructed over $R$, and $\Upsilon_i$ the set of all unfoldings of $G_i$ with instantiations $\beta : X_i \to \mathcal{T}_\Sigma$. The program body $t_i$ is the maximal pattern of all unfoldings in $\Upsilon_i$.*

*Let $X_i^R = \mathbf{var}(t_i[U_{rec} \leftarrow \Omega])$ be the set of variables occuring in the program body. Let $X_i^H = X_i \setminus X_i^R$ be the set of hidden variables of subprogram $G_i$. Let $\mathbf{sub}(x_j, r) \in \mathcal{T}_\Sigma(X_i)$ be a substitution term with $\mathbf{var}(\mathbf{sub}(x_j, r)) \cap X_i^H \neq \emptyset$. Let $x_h \in \mathbf{var}(\mathbf{sub}(s_j, r)) \cap X_i^H$ be a hidden variable used in the substitution term at position $u_h \in \mathbf{pos}(\mathbf{sub}(x_j, r), x_h)$. It holds:*

1. $\neg\exists x_k \in X_i^R$ *with* $\forall w \in W : \beta_{w \circ r}(x_j)|_{u_h} = \beta_w(x_k)$.
2. $\neg\forall w_1, w_2 \in W : \mathbf{node}(\beta_{w_1 \circ r}(x_j), u_h) = \mathbf{node}(\beta_{w_2 \circ r}(x_j), u_h)$.
3. $\neg\exists u \leq u_h$ *and* $\neg\exists x_k \in X_i$ *with* $\forall w \in W : \beta_{w \circ r}(x_j)|_u = \beta_w(x_k)$.

**Proof 7.3 (Identification of Hidden Variables)**

1. *Suppose there exists a variable $x_k \in X_i^R$ for which holds condition 1 in lemma 7.5, then it must hold that $x_k \neq x_h$ because $x_k \in X_i^R$, $x_h \in X_i^H$, and $X_i^R \cup X_i^H = \emptyset$; and it must hold for all $w \in W$ that*
   - $\beta_{w \circ r}(x_j)|_{u_h} = \beta_w(x_k)$ *(given postulation)*
   - $\beta_{w \circ r}(x_j)|_{u_h} = \beta_w(x_h)$ *(follows from supposition)*

   *Consequently, for all $w \in W$ must hold $\beta_w(x_k) = \beta_w(x_h)$ which contradicts the assumption that $t_i$ is a maximal pattern of all unfoldings over $\beta$.*
2. *For all $w \in W$ holds that $\beta_{w \circ r}(x_j)|_{u_h} = \beta_w(x_h)$. Because $t_i$ is a maximal pattern, variables do not have a common non-trivial pattern (a common prefix). Therefore, there must*

*Table 7.14.* Determining Hidden Variables

**Function:** $\mathbf{hiddenVar} : \mathbf{pos}(t_{init}) \times X \times R \to X'$

**Pre:** $u$ is a position in $\beta(x_j, w, e)$
$X = \{x_1, \ldots, x_m\}$ is the set of variables
$\beta(x_k, w, e)$ returns the instantiation of variable $x_k \in X$ for segment $w \in W(e)$
$R$ is the index set of recursion points

**Post:** new variable symbol $x_{m+1}$

**Side-effect:** $X$ is extended by $x_{m+1}$; $\beta$ is extended by the values of $x_{m+1}$; Break (and backtracking to calculating segmentations) if there are not enough instances of $x_{m+1}$

1. $m = |X|$
2. Generate new variable symbol $x_{m+1} \notin X$
3. $X = X \cup \{x_{m+1}\}$
4. **FORALL** $e \in E$ **DO**
5. **FORALL** $w \in W$ **DO**
6. **IF** $w \circ r \in W(e)$
7. **THEN** $\beta(x_{m+1}, w, e) = \beta(x_j, w \circ r, e)|_u$
8. **IF NOT**($\mathbf{enoughInstances}(x_{m+1})$) (see table 7.13)
9. **THEN break**
10. **return** $x_{m+1}$

*exist two positions* $w_1, w_2 \in W$ *such that* $\mathbf{node}(\beta_{w_1 \circ r}(x_h), \lambda) \neq \mathbf{node}(\beta_{w_2 \circ r}(x_h), \lambda)$ *and therefore also* $\mathbf{node}(\beta_{w_1 \circ r}(x_j), u_v) \neq \mathbf{node}(\beta_{w_2 \circ r}(x_j), u_v)$.

3. *Follows from Lemma B.3 (appendix B3).*

From lemma 7.5 follows that for each substitution term of a variable can be decided whether it is constructed using an function from $\Sigma$ or a hidden variable. The initial instantiations of such hidden variables can be induced from the substitution terms. An algorithm for determining hidden variables is given in table 7.14.

### 3.4.4 ALGORITHM

The core of the algorithm for calculating substitution terms can be obtained by extending definition 7.44 and is given in table 7.15. The algorithm makes use of functions $\mathbf{uniqueness}$ (table 7.11), $\mathbf{betaRecurrent}$ (table 7.12), and $\mathbf{hiddenVar}$ (table 7.14).

Integrating all components for calculating substitutions introduced in this section, we obtain the following algorithm:

1. Instantiate $X = \mathbf{var}(\hat{t}_G)$ and calculate the initial instantiation as defined in 7.45.

*Table 7.15.* Calculating Substitution Terms of a Variable in a Recursive Call

**Function:** $\mathbf{calcSub} : X \times R \times \mathbf{pos}(t_{init}) \rightarrow \mathcal{T}_\Sigma$

**Pre:** $E$ is the set of indices for $T_{init}$
$R$ is the index set of recursion points with $r \in R$
$W(e)$ ist the set of segment indices for $t_e \in T_{init}$
$X$ is the set of variables in $\hat{t}_G$ and $x_j, x_k \in X$
$\beta(x_k, w, e)$ returns the instantiation of variable $x_k \in X$ for segment $w \in W(e)$

**Post:** a valid substitution term $\mathbf{sub}(x_j, r)$ for variable $x_j \in X$ in the $r-th$ recursive call.

**Side-effect:** Break caused if a variable is not uniquely determined (caused by **uniqueness**) or if there are not enough instances for calculating the instantiation of a hidden variable (caused by **hidden**).

$\mathbf{calcSub}(x_j, r, u)$ =

1. **IF** $\mathbf{betaRecurrent}(x_j, x_k, r, u)$
2. **AND (Let** $t_\beta = \beta(x_j, w \circ r, e)$ for $e \in E, w \in W(e)$ and $\beta(x_j, w \circ r, e) \neq \perp$ **In** $\mathbf{uniqueness}(x_j, x_k, u, t_\beta)$**)**
3. **THEN** $x_k$
4. **ELSE IF** $\forall e \in E, w \in W : \mathbf{node}(\beta(x_j.w \circ r, e), u) = f$ with $\alpha(f) = n$
5. **THEN** $f(\mathbf{calcSub}(x_j, r, u \circ 1), \ldots, \mathbf{calcSub}(x_j, r, u \circ n))$
6. **ELSE** $\mathbf{hidden}(u, x_j, r)$

2. Determine whether there are enough succeeding substitution terms as proposed in lemma 7.3. If there are not enough terms, break and backtrack to calculating a new segmentation.

3. For each variable $x_j \in X$ and each recursive call $r \in R$ calculate the substitutions, starting at $u = \lambda$ using algorithm 7.15.

4. Check consistency of instantiations as proposed in lemma 7.4. If an inconsistency is detected, break and backtrack to calculating a new segmentation.

5. If $|T_{init}| \geq 2$ then check whether there exist at least two initial trees in $T_{init}$ with complete initial instantiations as proposed in 7.2 in section 3.3. If this is not the case, break and backtrack to calculating a segmentation.

6. Initially, instantiate $t_G = \hat{t}_G$. Instantiate $m = |X|$. Calculate indices for the variables in $X$ using $\{1, \ldots, m\}$. For each $u_r \in U_{rec}$ extend $t_G$ in the following way: $t_G = t_G[u_r \leftarrow G(\mathbf{sub}(x_1, r), \ldots \mathbf{sub}(x_m, r))]$.

7. Return $G(x_1, \ldots x_m) = t_G$ and for all initial trees $t_e$ return $\beta_e = \beta(x_i, \lambda, e)$ for all $x_i \in X$.

## 3.5 CONSTRUCTING AN RPS

### 3.5.1 PARAMETER INSTANTIATIONS FOR THE MAIN PROGRAM

When constructing an RPS $\mathcal{S}(\mathcal{G}, t_0)$ which recursively explains a set of initial trees $T_{init}$, for each $t_e \in T_{init}$ there must be calculated the initial instantiation of variables $\beta_e$ in $t_0^e$. This can be done by using slightly modified versions of definitions 7.42 (instantiations of variables in a hypothetical subprogram body, section 3.2) and 7.45 (instantiations of variables in a segment, defined for parameter instantiations in recursive calls, section 3.4).

For a recursive program scheme $\mathcal{S}(\mathcal{G}, t_0)$ each initial tree $t_e \in T_{init}$ implies an instantiation $\beta_e$ of variables in $t_0$:

**Definition 7.46 (Instantiation of $t_0$)** *Let $T_{init} \subset \mathcal{T}_{\Sigma\cup\{\Omega\}}$ be a set of initial trees indexed over $E$ and $\mathcal{S}(\mathcal{G}, t_0)$ the RPS which recursively explains $T_{init}$. Let $t_0^e$ be the main program for $t_e \in T_{init}$. The instantiations of $t_0$ for a tree $t_e$ can be constructed as*

$$\beta_e(x) = \begin{cases} t_0^e|_u & \exists u \in \mathbf{pos}(t_0, x) \\ & with\ u \in \mathbf{pos}(t_0^e)\ and\ t_0^e|_u \neq \bot \\ \bot & otherwise. \end{cases}$$

If there exists a recursive subprogram which explains all trees in $T_{init}$, then the variable instantiations in the main program which calls this subprograms can be calculated with respect to the subprogram.

**Definition 7.47 (Instantiation of $t_0$ wrt a Subprogram)** *Let $T_{init} \subset \mathcal{T}_{\Sigma\cup\{\Omega\}}$ be a set of initial trees indexed over $E$ and $G(x_1, \ldots, x_m) = t_G$ be a subprogram which recursively explains the trees $t_e \in T_{init}$ together with variable instantiations $\beta_e^G : \{x_1, \ldots, x_m\} \to \mathcal{T}_\Sigma \cup \{\bot\}$. The instantiations of $t_0$ for a tree $t_e$ can be constructed as*

$$\beta_e(x) = \begin{cases} \beta_e^G(G(x_1, \ldots, x_m))|_u & (\mathbf{a})\ \exists u \in \mathbf{pos}(t_0, x)\ with \\ & u \in \mathbf{pos}(\beta_e^G(G(x_1, \ldots, x_m)))\ and \\ & (\mathbf{b})\ \beta_e^G(G(x_1, \ldots, x_m))|_u \neq \bot \\ \bot & otherwise. \end{cases}$$

### 3.5.2 PUTTING ALL PARTS TOGETHER

Now we have all components for a system for inducing recursive explanations from a set of initial trees. The core of the system is to find a set of recursive subprograms. An overview of the induction of a subprogram is given in figure 7.12. Program *BuildSub* involves the following steps: (1) finding a valid recurrent segmentation $U_{rec}$ for a set of initial trees (section 3.1), (2) constructing a hypothesis about the program body by calculating the maximal

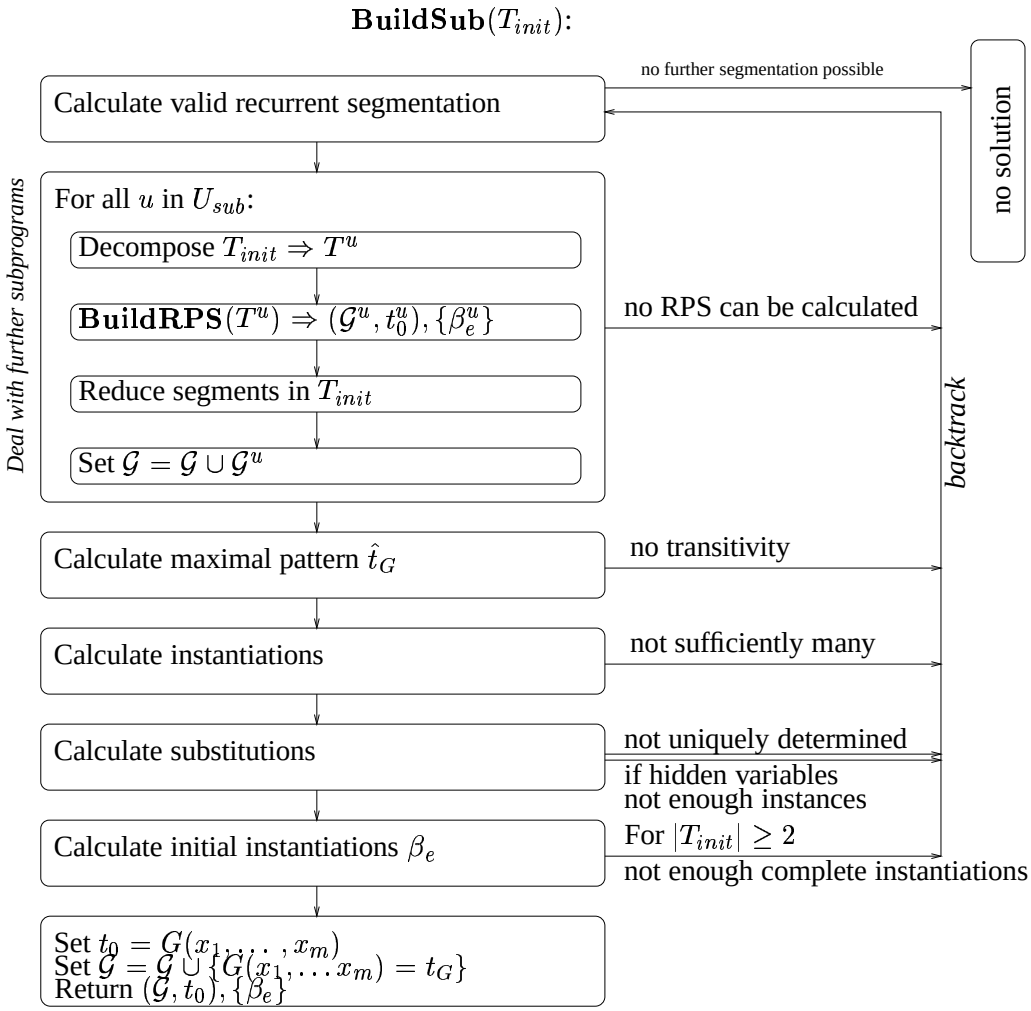



*Figure 7.12.* Steps for Calculating a Subprogram

pattern of the segments (section 3.2), (3) interpreting the remaining subtrees as variable instantiations, (4) determining the variable substitutions and the initial instantiation of the variables (section 3.4). If there are subtrees which are not covered by the current segmentation, these trees are considered to belong to further subprograms and the process of inducing a (sub-) RPS is called recursively (see *BuilRPS* in figure 7.13). Whenever one of the integrity conditions formulated in the sections above is violated, the algorithm backtracks to step one.

The complete procedure for inducing an RPS is given in figure 7.13. Starting at the roots of the initial trees in $T_{init}$, the system searches for a subprogram together with a calling main program and an initial instantiation of variables. Possibly, the searched for "top-level" subprogram calls further subprograms which are also constructed (*BuildSub*, figure 7.12). If no solution is found, the root node is interpreted as a constant function in the main program and *BuildRPS* starts with all subtrees of the root node as new set of initial trees. If none of the new initial trees contains an $\Omega$, no recursive generalization is constructed and there is only a main program which is the maximal pattern of all trees, otherwise, *BuildRPS* is called recursively for all sets of initial trees at fixed positions under the root.

The used strategy is to find a subprogram which explains as large a part of the initial trees as possible at a level in the trees as high as possible. Remember, that, if only one initial tree is given, it is not possible to determine the parameters of main program $t_0$, otherwise, the parameters of main are calculated by anti-unification of the main programs of all initial trees (see section 2.3). Remember further, that currently subprograms with different names are induced for each unexplained subtree of a to be induced subprogram. Subprograms with identical bodies can be identified after synthesis is completed (see section 3.3.4 and section 3.5.4).

### 3.5.3 EXISTENCE OF AN RPS

If an RPS can be induced from a set of initial trees using the approach presented here, then induction can be seen as a proof of existence of a recursive explanation – given the restrictions presented in section 2.2.

**Theorem 7.5 (Existence of an RPS)**
*Let $T_{init}$ be a set of initial trees indexed over $E$. $T_{init}$ can be explained recursively by an RPS iff:*

1. *$T_{init}$ can be recursively explained by a subprogram $G(x_1, \ldots, x_m) = t_G$, or*
2. *$\forall e \in E : \exists f \in \Sigma$ with $\alpha(f) = n$, $n > 0$, and $\mathbf{node}(t_e, \lambda) = f$, and $\forall T^k = \{t_e|_{k.\lambda} \mid k = 1, \ldots, n\}$ holds:*
   - *(a) $\forall t \in T^k : \mathbf{pos}(t, \Omega) = \emptyset$, or*
   - *(b) $\forall t \in T^k : \mathbf{pos}(t, \Omega) \neq \emptyset$ and it exists an RPS $\mathcal{S}^k = (\mathcal{G}^k, t_0^k)$ which recursively explains the trees in $T^k$.*

**Proof 7.4 (Existence of an RPS)**
*It must be shown (1) that an RPS exists, if propositions (1) and (2) in theorem 7.5 hold, and (2) that no RPS exists, if propositions (1) and (2) do not hold.*

1. **Existence** *of an RPS, if propositions (1) and (2) in theorem 7.5 hold:*

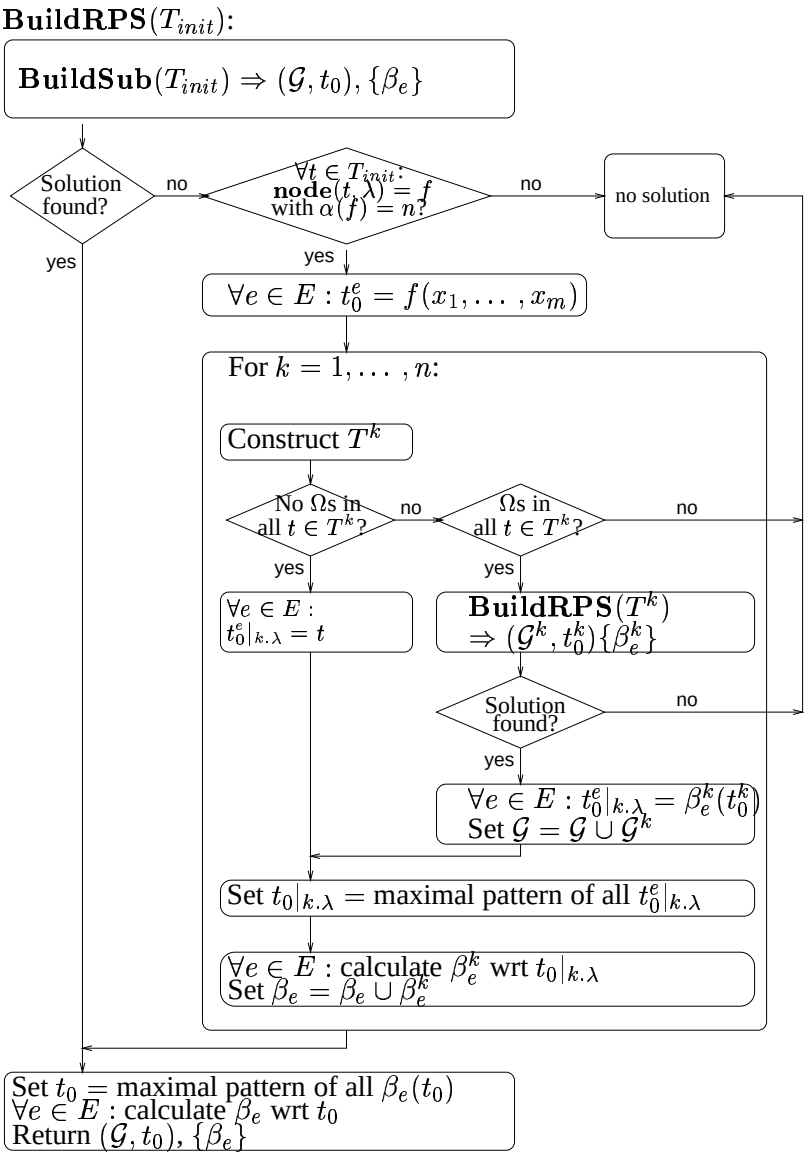



*Figure 7.13.* Overview of Inducing an RPS

**Proposition 1:** *Let $G(x_1, \ldots, x_m) = t_G$ be a subprogram which recursively explains all trees in $T_{init}$ (def. 7.30). Then there exist $\beta_e^G : \{x_1, \ldots, x_m\} \to \mathcal{T}_\Sigma \cup \{\bot\}$ such that $G$ with instantiations $\beta_e^G$ recursively explains $t_e \in T_{init}$. Let $t_0$ be the maximal pattern of all terms $\beta_e^G(G(x_1, \ldots, x_m))$ (theorem 7.1) and $\beta_e : \mathbf{var}(t_0) \to \mathcal{T}_\Sigma \cup \{\bot\}$ (def. 7.47) the initial instantiations of variables in $t_0$ for tree $t_e \in T_{init}$. Then the RPS*

$\mathcal{S}(\langle G(x_1, \ldots, x_m) = t_G \rangle, t_0)$ *together with instantiations* $\beta_e$ *recursively explains the trees in* $T_{init}$.

**Proposition 2:** *For all* $T^k$, $k = 1, \ldots, n$, *with* $\mathbf{pos}(t, \Omega) \neq emptyset$ *by assumption must exist an RPS* $\mathcal{S}^k = (\mathcal{G}^k, t_0^k)$ *which recursively explains trees* $T^k$ *and there must exist instantiations* $\beta_e^k : \mathbf{var}(t_0^k) \rightarrow \mathcal{T}_\Sigma \cup \{\bot\}$ *for parameters in the main programs* $t_0^k$. *For each term* $t_0^e$ *it must hold that* $\mathbf{node}(t_0^e, \lambda) = f$ *with arity* $\alpha(f) = n$. *For all* $k = 1, \ldots, n$ *must hold:*

$$t_0^e|_{k.\lambda} = \begin{cases} \beta_e^k(t_0^k) & \mathbf{pos}(t, \Omega) \neq \emptyset \\ t_e|_{k.\lambda} & \mathbf{pos}(t, \Omega) = \emptyset. \end{cases}$$

*The "global" main* $t_0$ *then is the maximal pattern of all* $t_0^e$ *with variable instantiations* $\beta_e(x)$ *(def. 7.46).* $\mathcal{G}$ *is the set of all subprograms of the sub-schemata* $\mathcal{S}^k$ *and the RPS* $\mathcal{S}(\mathcal{G}, t_0)$ *together with instantiations* $\beta_e$ *recursively explains all trees in* $T_{init}$.

2. **Non-Existence** *of an RPS, if propositions (1) and (2) in theorem 7.5 do not hold:*
*If proposition (1) do not hold, then there cannot exist an RPS with a main program* $t_0$ *which just calls a recursive subprogram. That is, there must exist an RPS with a larger main program for which proposition (2) does not hold.*
*If there does not exist a symbol* $f \in \Sigma$ *which is identical for all roots of the initial trees* $t_e \in T_{init}$, *then there cannot exist a (first-order) main program which explains all initial trees.*
*If there exist* $t_1 \in T^k$ *with* $\mathbf{pos}(t_1, \Omega) = \emptyset$ *and* $t_2 \in T^k$ *with* $\mathbf{pos}(t_2, \Omega) \neq \emptyset$, *then on the one hand,* $T^k$ *cannot be explained by an RPS (by assumption) because of* $t_1$, *and, on the other hand,* $T^k$ *cannot be explained by a non-recursive term with variable instantiations because of* $t_2$.

### 3.5.4 EXTENSION: EQUALITY OF SUBPROGRAMS

After an RPS $\mathcal{S}(\mathcal{G}, t_0)$ is induced, the set of subprograms $\mathcal{G}$ can possibly be reduced by unifying subprograms with equivalent bodies and different names $G_i \in \Phi$.[9] In the most simple case, the bodies of the two subprograms are identical except names of variables. In general, $G_i$ and $G_j$ can contain calls of further subprograms. $G_i$ and $G_j$ are equivalent, if these subprograms are equivalent, too. Table 7.16 gives an example, where $G_A$ calls a further subprogram $G_B$ and where variables of $G_A$ are partially instantiated – with different values for two different positions. Subprogram $G_A$ could be generated from the two partially instantiated subprograms by (a) determining whether the further subprograms are equivalent and by (b) constructing the maximal pattern of instances of $G_A$. But note, that a strategy for deciding when different subprograms should be identified as a single subprogram would be needed to realize this simplification of program schemes!

[9] This is a possible extension of the system, not implemented yet.

*Table 7.16.* Equality of Subprograms

$G_A(x_1, x_2, x_3) = if(eq1(x_1), G_B(x_2), G_A(-(x_1, x_3), x_2, x_3))$
Induced subprograms:
$G_A$ is called with $x_2 = 1$ and $x_3 = 2$:
$G_{A1}(x_1) = if(eq1(x_1), G_{B1}(1), G_{A1}(-(x_1, 2)))$

$G_A$ is called with $x_2 = 2$ and $x_3 = 3$:
$G_{A2}(x_1) = if(eq1(x_1), G_{B2}(2), G_{A2}(-(x_1, 3)))$

### 3.5.5 FAULT TOLERANCE

The current, purely syntactical approach to folding does allow for initial trees where some of the segments of a hypothetical unfolding are incomplete. Incompleteness could for example result from not considered cases when the initial trees were generated (by a user or another system). But, it does *not* allow for initial trees which contain defects, for example wrong function symbols. Such defects could for example result from user generated initial trees or traces, where it might easily happen that trees are unintentionally flawed (by typing errors etc.). A possible solution for this problem could be, that not only an RPS is induced but that additionally, minimal transformations of the given initial trees are calculated and proposed to the user. This problem is open for further research.

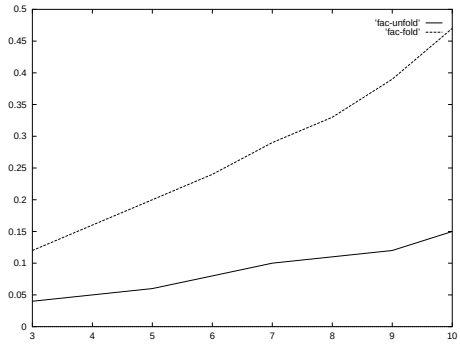
## 4 EXAMPLE PROBLEMS

To illustrate the presented approach to folding finite programs we present some examples. First, we give time measures for some standard problems. Afterwards we demonstrate application to planning problems which we introduced in chapter 3.

## 4.1 TIME EFFORT OF FOLDING

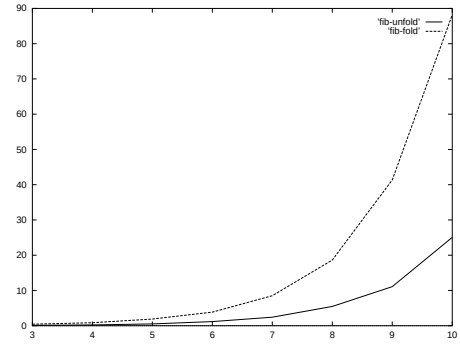
Figures 7.14 and 7.15 give the time effort for unfolding and folding the factorial (see tab. 7.7) and the Fibonacci function (see tab. 7.3). The experiments were done by unfolding a given RPS to a certain depth and then folding the resulting finite program term again. In all experiments the original RPS could be inferred from its $n$-th unfolding ($n = 3, \ldots, 10$). The procedure for obtaining the time measures is described in appendix A6.

Folding has a time effort which is a factor between 3 and 4 higher than unfolding. For linear recursive functions, folding time is linear and for tree recursive functions, folding time is exponential in the number of unfolding points, as should be expected.




Unfolding-depth $n = 3, \ldots, 10$ ($x$-axis) measured in seconds ($y$-axis)

*Figure 7.14.* Time Effort for Unfolding/Folding *Factorial*



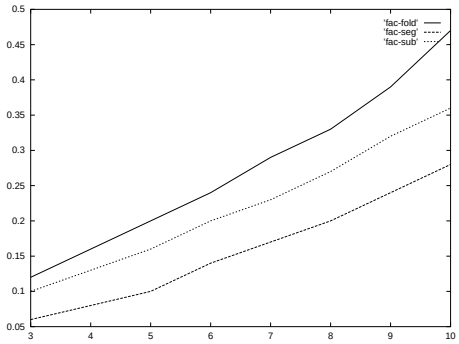

Unfolding-depth $n = 3, \ldots, 10$ ($x$-axis) measured in seconds ($y$-axis)

*Figure 7.15.* Time Effort for Unfolding/Folding *Fibonacci*

The greatest amount of folding time is used for constructing the substitutions and the second greatest amount is used for calculating a valid recurrent segmentation. Figure 7.16 gives these times for the factorial function.

Both for *factorial* and for *Fibonacci* the first segmentation hypothesis leads to success, that is, no backtracking is needed. For a variant of *factorial* with a constant initial part (see table 7.17), one backtrack is needed. For an unfolding depth $n = 3$ time for folding goes up to $0.36\ sec$ (from $0.12\ sec$) and for $n = 4$ to $0.44\ sec$ (from $0.16\ sec$).

For the *ModList* function (see tab. 7.2) an RPS with two subprograms must be inferred. For the third unfolding of *ModList* where *Mod* is unfolded once in the first, twice in the second, and three times in the third unfolding is $1.07\ sec$ (with $0.16\ sec$ for unfolding). For the fourth unfolding with one to four unfoldings of *Mod*, folding time is $3.20\ sec$ (with $0.33\ sec$ for unfolding).




Unfolding-depth $n = 3, \ldots, 10$ ($x$-axis) measured in seconds ($y$-axis)

*Figure 7.16.* Time Effort Calculating Valid Recurrent Segmentations and Substitutions for *Factorial*

*Table 7.17.* RPS for *Factorial* with Constant Expression in *Main*

$\mathcal{G} = \langle G(x) = if(eq0(pred(x)), 1, *(x, G(pred(x))))\rangle$
$t_0 = if(eq0(x), 1, G(x))$

## 4.2 RECURSIVE CONTROL RULES

In the following we present how RPSs for planning problems such as the ones presented in chapter 3 can be induced. In this section we do *not* describe how appropriate initial trees can be generated from universal plans but just demonstrate that the recursive rules underlying a given planning domain can be induced if appropriate initial trees are given!

For the function symbols used in the signature of the RPSs we assume that they are predefined. Typically such symbols refer to predicates and operators defined in the planning domain. Furthermore, for planning problems, a situation variable $s$ is introduced which represents the current planning state. A state can be presented as list of literals – for Strips-like planning (see chap. 2) – or as list or array over primitive types such as numbers – for a functional extension of Strips (see chap. 4). Generating initial trees from universal plans is discussed in detail in chapter 8.

The *clearblock* problem introduced in section 1.4.1 in chapter 3 has an underlying linear recursive structure. In figure 7.17 an initial tree for a three block problem is given (the *ontable* literal is omitted for better readability). The resulting RPS is:

$$\mathcal{G} = \langle G(x, s) = if(clear(x, s), s, puttable(topof(x, s), G(topof(x, s), s)))\rangle$$

$$t_0 = G(C, list(clear(A), on(A, B), on(B, C), ontable(C))).$$

The first variable $x$ holds the name of the block to be cleared, for example $C$. The situation variable $s$ holds a description of a planning state as list of literals. Predicate *clear*, selector *topof*, and operator *puttable* are assumed to be predefined functions. *Clear(x, s)* is true if $clear(x) \in s$; *topof(x, s)* returns $y$ for $on(y, x) \in s$; *puttable(x, s)* deletes *on(x, y)* from $s$ and adds the literals *clear(y)* and *ontable(x)* to $s$.

The Tower of Hanoi domain is an example for a recursive function relying on a hidden variable (see sect. 3.4.3). In figure 7.18, an initial tree is given. The underlying RPS is:

$$\mathcal{G} = \langle \quad G(n,x,y,z) = if(eq1(n), move(x,y), \\ ap(hanoi(pred(n),x,z,y), move(x,y), hanoi(pred(n),z,y,x)))\rangle$$

$$t_0 = G(3,a,b,c).$$

The function symbol *ap* (which can be interpreted as *append*) is used to combine two calls of *hanoi* with a *move* operation.

Parameter $z$ does occur only in substitutions in the recursive calls. Therefore, the maximal pattern contains only three variables:

$$\hat{t}_G = if( \quad eq1(x_1), \\ move(x_2,x_3), \\ ap(G, move(x_2,x_3), G))$$

and the fourth variable $z$ is included after determining the substitution terms.

In the following chapter, recursive generalizations for these and further planning problems will be discussed.

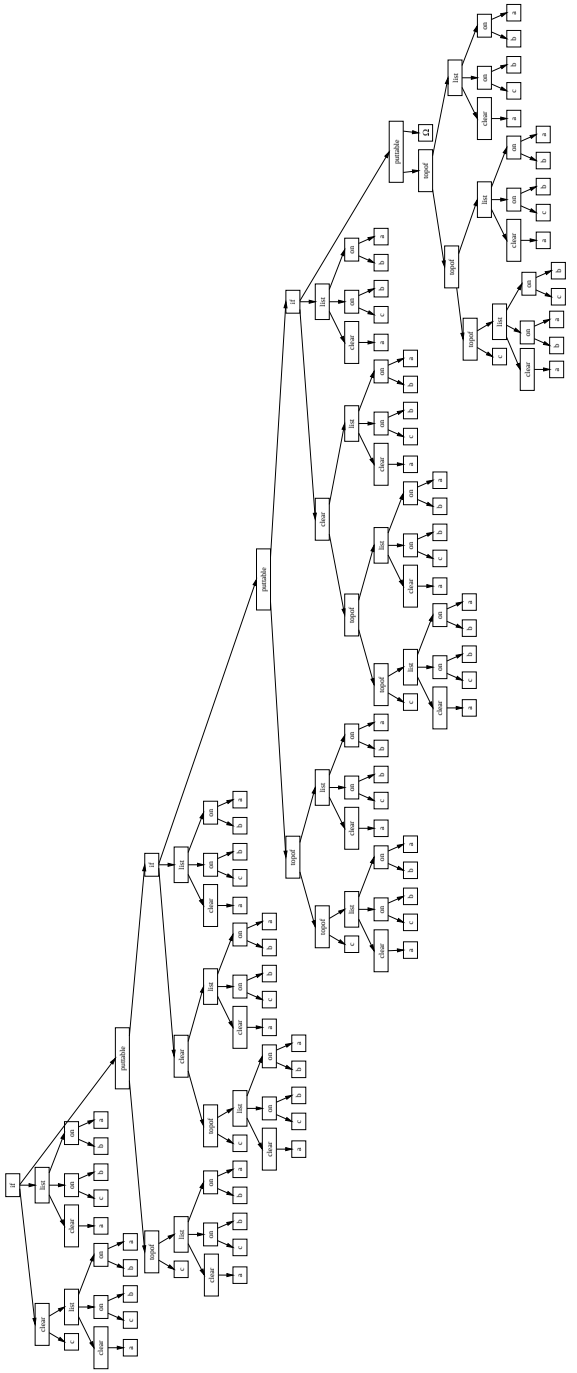


*Figure 7.17.* Initial Tree for *Clearblock*

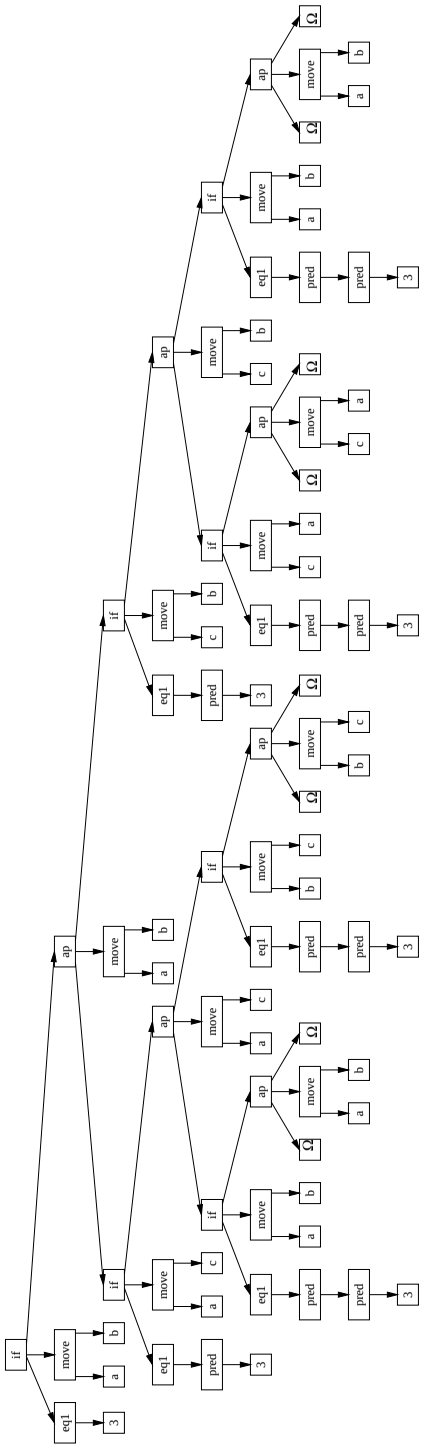



*Figure 7.18.* Initial Tree for *Tower of Hanoi*

Chapter 8

# TRANSFORMING PLANS INTO FINITE PROGRAMS

*"[...] I guess I can put two and two together." "Sometimes the answer's four," I said, "and sometimes it's twenty-two. [...]"*

—Nick Charles to Morelli in: Dashiell Hammett, *The Thin Man*, 1932

Now we are ready to combine universal planning as described in chapter 3 and induction of recursive program schemes as described in chapter 7. In this chapter, we introduce an approach to transform plans generated by universal planning into finite programs which are used as input to our folder. On the one hand, we present an alternative approach to realizing the first step of inductive program synthesis as described in section 3.4 in chapter 6 – using AI planning as basis for generating program traces. On the other hand, we demonstrate that inductive program synthesis can be applied to the generation of recursive control rules for planning – as discussed in section 5.2 in chapter 2. As a reminder to our overall goal introduced in chapter 1, in figure 8.1 an overview of learning recursive rules from some initial planning experience is given.

While plan construction and folding 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. What we present here is work in progress. As a consequence, the algorithms presented in this chapter are stated rather informally or given as part of the *Lisp* code of the current implementation and we rely heavily on examples. We can demonstrate that our basic idea – transformation based on data type inference – works for a variety of domains and hope to elaborate and formalize plan transformation in the near future.

In the following, we first give an overview of our approach (sect. 1), then we introduce decomposition, type inference, and introduction of situation variables

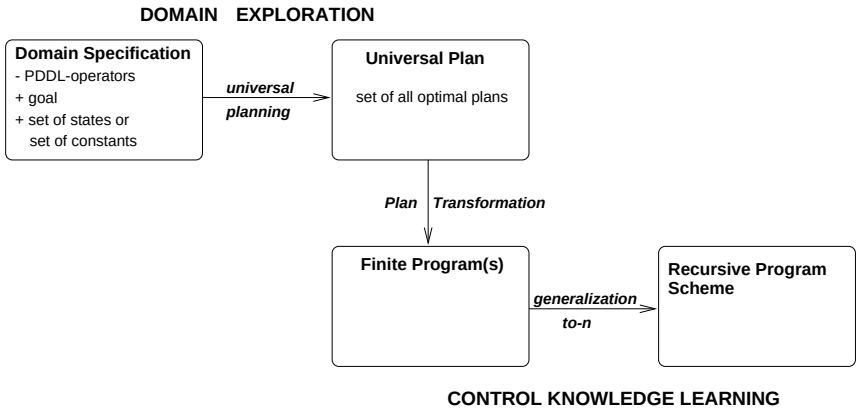



*Figure 8.1.* Induction of Recursive Functions from Plans

as components of plan transformation (sect. 2). In the remaining sections we give examples for plans over totally ordered sequences (sect. 3), sets (sect. 4), partially ordered lists (sect. 5), and more complex types (sect. 6).[1]

# 1 OVERVIEW OF PLAN TRANSFORMATION

## 1.1 UNIVERSAL PLANS

Remember construction of universal plans for a small finite set of states and a fixed goal, as introduced in chapter 3. A universal plan is a DAG which represents an optimal, *totally ordered sequence* of actions (operations) to transform each input state considered in the plan into the desired output (goal) state. Each state is a node in the plan and is typically represented as set of atoms over domain objects (constants). Actions in a plan are applied to the current state in working memory. For example, for a given state $\{$*on(A, B), on(B, C), clear(A), ontable(C)*$\}$, application of action *puttable(A)* results in $\{$*on(B, C), clear(A), clear(B), ontable(A), ontable(C)*$\}$.

## 1.2 INTRODUCING DATA TYPES AND SITUATION VARIABLES

To transform a plan into a finite program, it is necessary to introduce an order over the domain (which can be gained by introducing a data type) and to introduce a variable which holds the current state (i. e., a situation variable).

[1] This chapter is based on the previous publications Schmid and Wysotzki (2000b) and Schmid and Wysotzki (2000a). An approach which shares some aspects with the plan transformation presented here and is based on the assumption of linearizability of goals is presented in Wysotzki and Schmid (2001).

*Data type inference* is crucial for plan transformation: The universal plan already represents the structure of the searched-for program, but it does not contain information about the *order* of the objects of its domain. Typically for planning, domain objects are represented as sets of constants. For example, in the *clearblock* problem (sect. 1.4.1 in chap. 3) we have blocks *A*, *B*, and *C*. As discussed in chapter 6, approaches to deductive as well as inductive functional program synthesis rely on a "constructive" representation of the domain. For example, Manna and Waldinger (1987) introduce the *hat*-axiom for refering to a block which is lying on another block; Summers (1977) relies on a predefined complete partial order over lists. Our aim is, to *infer* the data structure for a given planning domain. If the data type is known, constant objects in the plan can be replaced by constructive expressions.

Furthermore, to transform a plan into a program term, a *situation variable* (see sect. 2.2 in chap. 2 and sect. 2.1.2 in chap. 6) must be introduced. While a planning algorithm applies an instantiated operator to the current state in working memory (that is, a "global" variable), interpretation of a functional program depends only on the instantiated parameters (that is, "local" variables) of this program term (Field and Harrison, 1988). Introducing a situation variable in the plan makes it possible to treat a state as valuated parameter of an expression. That is, the primitive operators are now applied to the set of literals held by the situation variable.

## 1.3 COMPONENTS OF PLAN TRANSFORMATION

Overall, plan transformation consists of three steps, which we will describe in the following section:

**Plan Decomposition:** If a universal plan consists of parts with different sets of action names (e. g., a part where only *unload(x, y)* is used and another part, where only *load(x, y)* is used), 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.

For information about the global data structures and central components of the algorithm, see appendix A7.

## 1.4 PLANS AS PROGRAMS

As stated above, a universal plan represents the optimal transformation sequences for each state of the finite problem domain for which the plan was constructed. To execute such 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 term, this program can be evaluated 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 input 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.*[2] Our goal is to extract the underlying program structure from the plan. To interpret the plan as a (functional) program term, states are re-interpreted as boolean operators: All literals of a state description which are involved in transformation – the "footprint" (Veloso, 1994) – are rewritten with the predicate symbol as boolean operator introducing a situation variable as additional argument. Each boolean operator is rewritten as a function which returns *true*, if some proposition holds in the current situation, and *false* otherwise. Additionally, the actions are extended by a situation variable, thus, the current (partial) description of a situation can be passed through the transformations. Finally, to represent the conditional statement given above, 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 8.1 (Plan as Program)** *Each node $S$ (set of literals) in plan $\Theta$ is interpreted as conjunction $B$ of boolean expressions. The planning tree can now be interpreted as nested conditional: $\Theta(s) =$ 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*

[2] An interpreter function for universal plans is given in (Wysotzki and Schmid, 2001).

*at the edge from $B$ to a child node, and $\Theta'$ 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* $x_n\ t_n\ \Omega$*))))).*

In general, boolean expressions $B(s)$ can involve propositions explicitely given in a situation as well as additional constraints as described in chapter 4. 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 some problems a complex structure can be collapsed into a linear one as a result of data type introduction.

## 1.5 COMPLETENESS AND CORRECTNESS

Starting point for plan transformation is a *complete* and *correct* universal plan (see chap. 3). 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 folding 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. For example, if a plan for sorting lists of up to three elements was generalized into a recursive sorting program, the user could test this program by presenting lists with for or more elements as input and inspecting the resulting output.

# 2 TRANSFORMATION AND TYPE INFERENCE

## 2.1 PLAN DECOMPOSITION

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

**Definition 8.2 (Uniform Sub-Plan)** *A sub-plan is uniform if it contains only fixed, regular sequences of operator-names* $\langle o_1 \ldots o_n \rangle$ *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. 8.2.a). An example for such a plan is the *clearblock* plan given in figure 3.2 which consists of a linear sequence of *puttable* actions. It could also be possible that some operators are applied in a regular way – for example *drill-hole–polish-object* (see fig. 8.2.b).

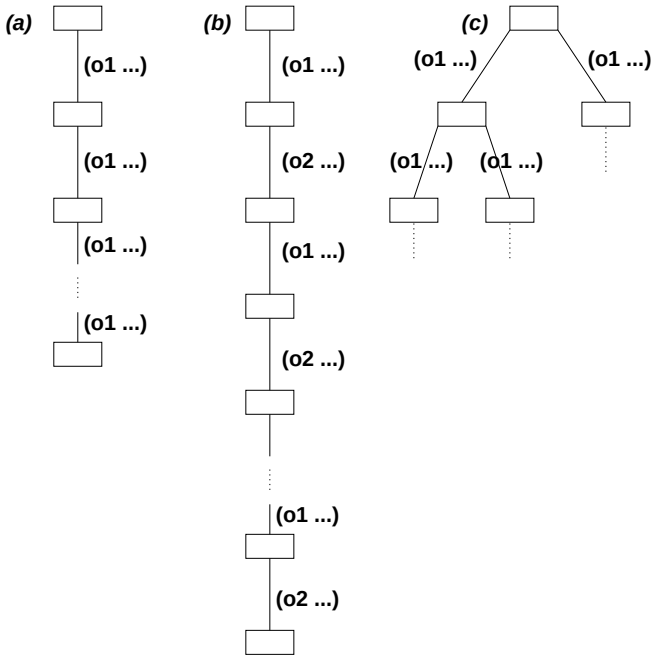



*Figure 8.2.* Examples of Uniform Sub-Plans

Single operators or regular sequences can alternatively occur in more complex planning structures (see fig. 8.2.c). An example for such a plan is the *sorting* plan given in figure 3.7 which is a DAG where each arc is labelled with a *swap* action.

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. 8.3.a). This is, for example, the case for the *rocket* domain (see plan in fig. 3.5): The first levels of the DAG only contain *unload* actions, followed by a single *move-rocket* action, followed by some levels which only contain *load* actions. In general, sub-plans can occur at any position in the planning structure as subgraphs (see fig. 8.3.b).

We have only implemented a very restricted mode for this initial plan decomposition: single operators and level-wise splitting (see appendix A8). A full implementation of decomposition involves complex pattern-matching, as it is realized for identifying sub-programs in our folder (chap. 7). 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, that is, it terminates with more than one leaf. Each leaf becomes the root of a potential subsequent sub-plan.

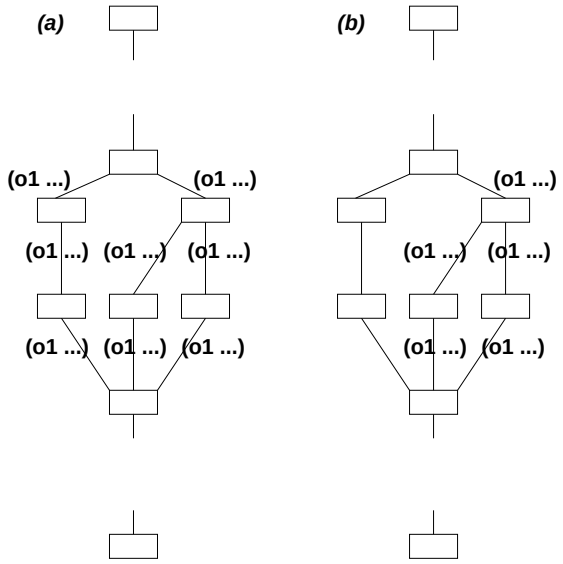



*Figure 8.3.* Uniform Plans as Subgraphs

In the current implementation, we return the complete plan if different operators occur at the same level. A reasonable minimal extension (still avoiding complex pattern-matching as described in chap. 7) would be to search for subplans fulfilling our simple splitting criterium at lower levels of the plan. But up to now, this case only occurred for complex list problems (such as *tower* with 4 blocks, see 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 (automatically generated) names of the sub-plans (see also chap. 7), which are initially associated with the partial plans and finally with recursive functions. If folding 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 with name $p_i$ is a situation variable. For all arguments $arg_i$, the initial values as given in the finite program are known – as a result from folding where parameters and their initial instantiations are identified in an initial program (chap. 7).

## 2.2 DATA TYPE INFERENCE

The central step of plan transformation is data type inference.[3] The structure of a (sub-) plan is used to generate an hypothesis about the underlying 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 8.3 (Data Type)** *A data type $\tau$ 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 $\perp < c(x, \perp) < c(x', c(x, \perp)) \ldots$. If $x$ is empty or unique, the data type is simple and the (countable, infinite) set of elements belonging to $\tau$ 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.*

An example for a data type is *list* with the empty list as bottom element, *cons(x, l)* as list-constructor, *head(l)* as selector for the first element of a list, and *tail(l)* as selector for the "rest" of the list, that is, the list without the first element.

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

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

way. Referring to objects by functions $f(t)$, where $t$ is a ground term (a constant, such 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, 2000). For example, *(pick oset)* can represent a specific object in an object list of arbitrary length.

## 2.3 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 8.1. An abbreviated version of the rewriting-algorithm is given in appendix A9.[4]

# 3 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 must exist 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.

**Definition 8.4 (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 the algorithm given in table 8.1. Note, that inference is performed over a plan which is generated by our Lisp-implemented planning system. Therefore, expressions are represented within parenthesis. For example, the literal, representing that a block $A$ is lying on another block $B$ is represented as *(on A B)*; the action to put block $A$ on the table is represented as *(puttable A)*. The restriction to a single top-level goal in the algorithm can be extended to multiple goals by a slight modification.[5]

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 8.1. The program code for generating this function

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

[5]Data type *sequence* must be introduced as first step. If the sequence of objects in the plan is identified, it is also identified which predicate characterizes the elements of this sequence. The top-level goals must include this predicate with the bottom-element as argument and consequently can be selected as "bottom-test" predicate.

*Table 8.1.* 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 $(g\ a_1 \ldots a_n)$ set **bottom-test** to $p$, else fail.
    - 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 $a_1$, keep it
      otherwise, remove all arguments $a_i$ 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 $((e_1(succe_0)) \ldots (e_m(succ^{m-1}e_0)))$.
      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 $(pose_i, i+1, q) = (pos\ e_0\ 1\ q) = p_i$ 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 $(g\ a_1 \ldots a_n)$
      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.
- If the plan starts at a level $> 0$:an "intermediate" goal must be identified; afterwards, proceed as above.

is given in figure 8.4. The first function (*succ-pattern*) represents a pre-defined pattern for the successor-function. For a concrete plan, this pattern must be instantiated in accordance with the predicates occuring in the plan. For example, for the *clearblock* problem (see chap. 3 and below), the successor function *topof(x) = y* is constructed from predicates *on(y, x)*.

**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. Such a planning domain was specified by *clearblock* as presented in section 1.4.1 in chapter 3. Here we use a slightly different domain, omitting the predicate *ontable* and with operator *puttable* named *unstack*. The domain specification and a plan for *unstack* are given in figure 8.5.

The protocol of plan-transformation is given in figure 8.6. 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 pre-defined (Manna and Waldinger, 1987; Geffner, 2000; Wysotzki and Schmid, 2001), we can infer them from the

```
; 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)
(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 8.4.* Generating the *Successor*-Function for a Sequence

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

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

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

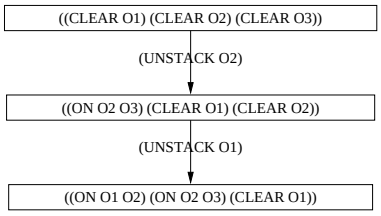



*Figure 8.5.* The *Unstack* Domain and Plan

universal plan. The "constructively" rewritten plan where data type *sequence* is introduced is given in figure 8.7. The transformation information stored for the finite program is given in appendix C3.

The finite program term which can be constructed from the term given in figure 8.7 is:

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

```
+++++++++ Transform Plan to Program  ++++++++++
1st step: decompose by operator-type
Single Plan
(SAVE SUBPLAN 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))))
-----------------------------------------------
3rd step: Transform plan to program
Show Plan as Program? y
-------------------------------------------------------
```

*Figure 8.6.* Protocol for *Unstack*

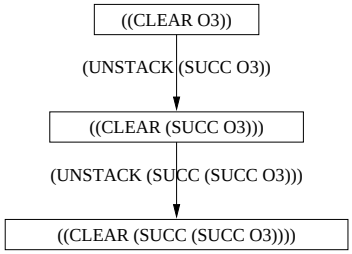



*Figure 8.7.* Introduction of Data Type *Sequence* in *Unstack*

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

An RPS (see chap. 7) generalizing this *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 $t_0$ =

```
; 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 8.8.* LISP-Program for *Unstack*

*(unstack-rec o s)* for some constant *o* and some set of literals $s$. The executable program is given in figure 8.8.

Plans consisting of a linear sequence of operator applications over a sequence of objects in general result in generalized control knowledge in form 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 8.2.

It is simple and straight-forward to generalize over linear plans. For this plans of problems, our approach automatically 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!

*Table 8.2.* 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))) |

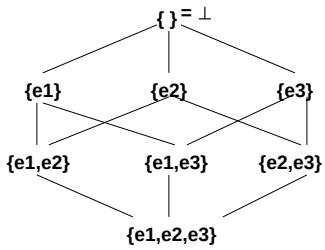



*Figure 8.9.* Partial Order of *Set*

## 4 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 8.9 – this order corresponds to the sub-plans for *unload* or *load* in the *rocket* domain (sect. 1.4.2 in chap. 3). If *pick* and *rst* are defined deterministically (e. g., by list-selectors), the partial order gets reduced to a total order.

**Definition 8.5 (Set)** *Data type set is defined as:*
$set = \bot \mid c(e, set)$ *with*

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

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

*Table 8.3.* Introducing *Set*

- Collapse plan to one path (implemented as: take the "leftmost" path).
- Generate a *complex data object*: like *Generate Sequence* in tab. 8.1.
  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 $(rst set)$ are predefined by $car$ and $cdr$.

Because the complex data object is inferred from the top-level goals (given 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 the algorithm given in table 8.3. Because we collapse such a plan to one path of a set of paths, this algorithm completely contains the case of sequences (tab. 8.1). Note, that collapsing plans with underlying data type set corresponds to the idea of "commutative pruning" as discussed in (Haslum and Geffner, 2000).
The program code for `make-co`, generating the bottom-test, and for `pick`, and `rst` referred to in algorithm 8.3 is given in figure 8.10. As described for data type sequence above, the *pattern* of the selector functions are pre-defined and the concrete functions are constructed by instantiating these patterns with information gained from the given plan. Introduction of a new predicate with a complex argument corresponds to the notion of "predicate-invention" in Wysotzki and Schmid (2001).[6]

**Oneway Transportation.** A typical example for a domain with a set as underlying data type are simple transportation domains, where some objects must be loaded into a vehicle which moves them to a new location. Such a domain is *rocket* (see sect. 1.4.2 in chap. 3). The *rocket* plan is in a first step levelwise decomposed into sub-plans with uniform actions (see fig. 8.11).

[6]This kind of predicate invention should not be confused with predicate invention in ILP (see sect. 3.3 in chap. 6). In ILP a new predicate must necessarily be introduced if the current hypothesis covers some negative examples.

```
; 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)
(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 8.10.* Functions Inferred/Provided for Set

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

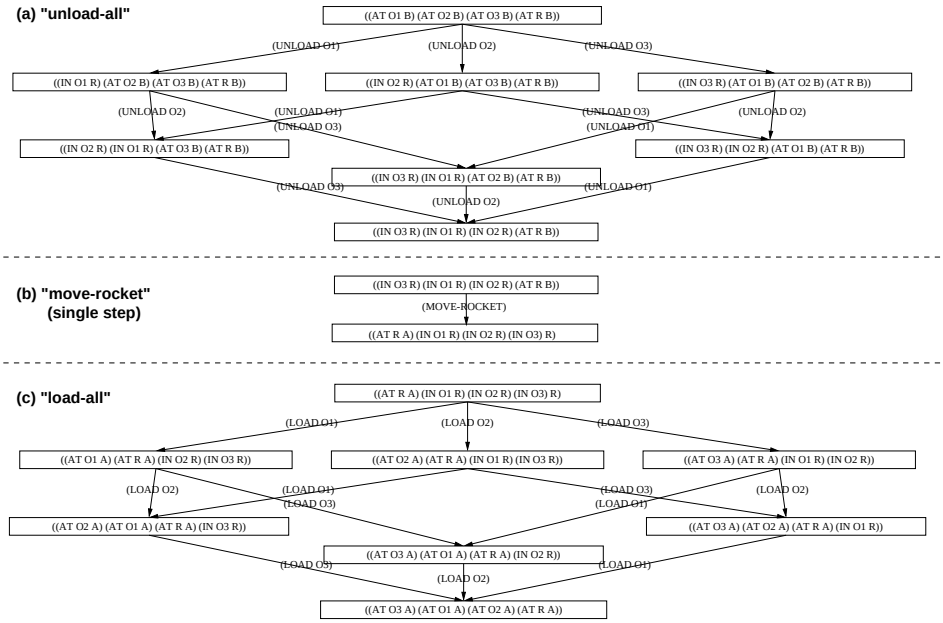



*Figure 8.11.* Sub-Plans of *Rocket*

performed is irrelevant, that is, the underlying data structure of both sub-plans is a *set*. Consequently, the (sub-) plan can be collapsed to one path.

Introduction of the data type set involves the following steps:

- 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** of the 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*

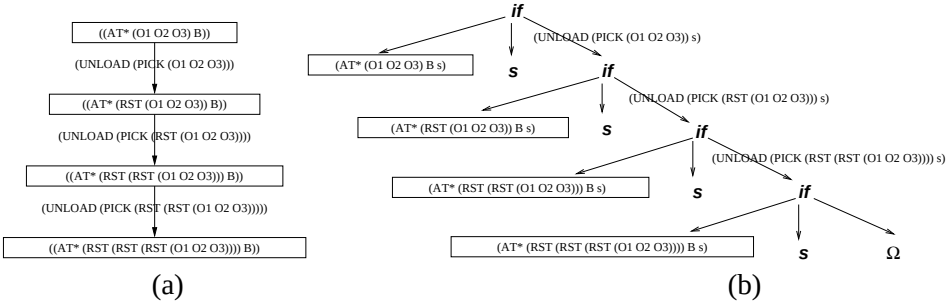



*Figure 8.12.* Introduction of the Data Type *Set* (a) and Resulting Finite Program (b) for the *Unload-All* Sub-Plan of *Rocket* ($\Omega$ denotes "undefined")

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 $rst$ deterministically (e. g., as *head*/*tail* or *last*/*butlast*). Although it is irrelevant which object is selected next, it is necessary that *rst* returns exactly the set of objects without the currently picked one.

The transformed sub-plan for *unload-all* is given in figure 8.12.a. 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. Now the plan can be read as a nested conditional expression (see fig. 8.12.b).

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

```
(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 folding 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.

The protocol of plan-transformation is given in figure 8.13.

```
+++++++++ 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)))
-----------------------------------------------
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]
-----------------------------------------------
3rd step: Transform plan to program
Show Plan(s) as Program(s)? y
```

*Figure 8.13.* Protocol of Transforming the *Rocket* Plan

Recursive functions for unloading and loading objects are:

$$
\begin{aligned}
unload-all(oset, s) = if( \quad & at*(oset, B, s), \\
& s, \\
& unload(pick(oset), unload-all(rst(oset), s))) \\
load-all(oset, s) = if( \quad & inside*(oset, Rocket, s), \\
& s, \\
& load(pick(oset), load-all(rst(oset), s)))
\end{aligned}
$$

which are integrated in the "main" program `(rocket oset s)` given above.

The recursive functions for loading and unloading all objects in some arbitrary order learned from the object *rocket* problem 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 *load-all* function must involve an additional condition:

$$\begin{aligned} load-all(oset,c,s) = if( \quad & or(eq0(c), at*(oset,B,s)),\\ & s,\\ & load(pick(oset), load-all(rst(oset), pred(c), s))). \end{aligned}$$

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 C4 together with a short discussion about interleaving the *inside* and *at* predicates.

# 5 PLANS OVER LISTS OF OBJECTS

## 5.1 STRUCTURAL AND SEMANTIC 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*

[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, see chap. 4. Currently we cannot generate disjunctive boolean operators in plan transformation.

*Table 8.4.* 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<br>(unload (pick oset) (rec-unload (rst oset) s))) |
| (c) | (filter p l) == | (if (empty l) nil<br>(if (p (head l)) (cons (head l) (filter p (tail l))) (filter p (tail l)))) |
| | (odd-els l) == | (if (empty l) nil<br>(if (odd (head l)) (cons (head l) (filter (tail l))) (filter (tail l)))) |
| | (member e l) == | (if (empty l) nil<br>(if (equal (head l) e) e (member e (tail l)))) |

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 8.4 illustrates structural list-problems.

Structural list problems, as discussed for example in section 3.4.1 in chapter 6, can be dealt with by an algorithm nearly identical to algorithm in table 8.3 dealing with sets (see tab. 8.5). Because the only relevant information is the *length* of a list, the partial order can be reduced to a total order (see fig. 8.14). Generating a total order results in linearizing the problem (Wysotzki and Schmid, 2001). The extraction of a unique path in the plan is only slightly more complicated as for sets and is discussed below.

**Definition 8.6 (List)** *Data type list is defined as:*
$list = nil \mid cons(e, list)$ *with*

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

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

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

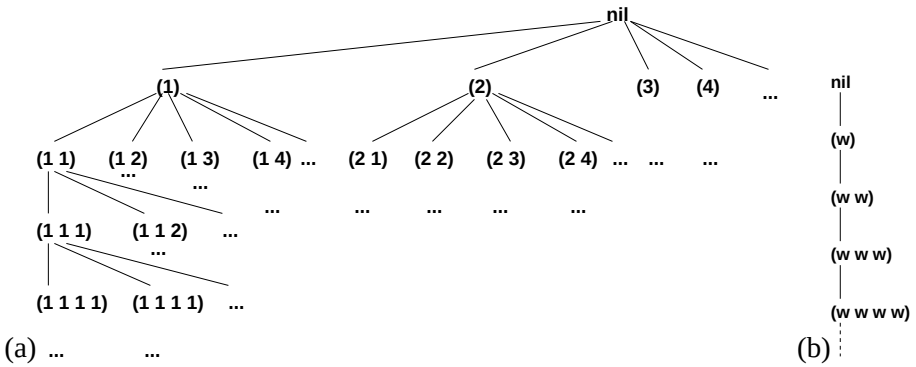



*Figure 8.14.* Partial Order (a) and Total Order (b) of Flat Lists over Numbers

*Table 8.5.* Introducing *List*

- *Collapse plan to one path. *(discussed in detail below)*
- 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.

and Yilmaz, 1999; Le Blanc, 1994) and inductive functional program synthesis typically is restricted to structural problems (Summers, 1977).

Currently, we approach transformation for such problems by the steps presented in the algorithm in table 8.6. 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.

*Table 8.6.* 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 *semantic* 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 8.5.

## 5.2 SYNTHESIZING 'SELECTION-SORT'

### 5.2.1 A PLAN FOR SORTING LISTS

The specification for sorting lists with four elements is given in table 3.6. 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 (such 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 figure 4.10. This second version allows for plan construction without using a set of predefined states. Information about 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 neighboring 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 3.7. For sorting a list of four elements, there exist 24 states. Swapping elements with the restrictions given for selection sort results in 72 edges. Sorting lists of three elements is illustrated in appendix C5.

### 5.2.2 DIFFERENT REALIZATIONS OF 'SELECTION SORT'

To make the transformation steps more intuitive, we first discuss functional variants of *selsort* (see tab. 8.7): 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. 8.1). The second variant of selection sort is a linear recursion: 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 $c - 1$. The 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 3.7. 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*! That is, it does not rely on the static predicate *gt(x, y)* which represents the order relation of the first $n$ natural numbers (see fig. 3.6).

### 5.2.3 INFERRING THE FUNCTION-SKELETON

The plan given in figure 3.7 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. 2 in chap. 3).

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

*Table 8.7.* 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)
)
```

The plan for sorting lists with three elements (see appendix C5) 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 have no 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

*Table 8.8.* Extract an MST from a DAG

- Input: a DAG with edges $(n\ m)$ 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 $c \in Cart$.

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 A10 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 8.15. For bettern readability, the state descriptions in the nodes are given as lists. In the original plan, each list is described by a set of literals. For example, $[1\ 2\ 3\ 4]$ is a short notation for *(isc p1 1) (isc p2 2) (isc p3 3) (isc p4 4)* where *isc(x y)* means that position $x$ has content $y$ (see sect. 1.4.3 in chap. 3 for details).

The algorithm for extracting a minimal spanning tree from a DAG is given in table 8.8, the corresponding program fragment in appendix A11.

The original plan represented how each of the 24 possible lists over four (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 state. In the minimal spanning tree given in figure 8.15, the action sequences for sorting lists by shifting the smallest element to the left is given.

For collapsing a tree into a single path, this tree has to be regular[10]:

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

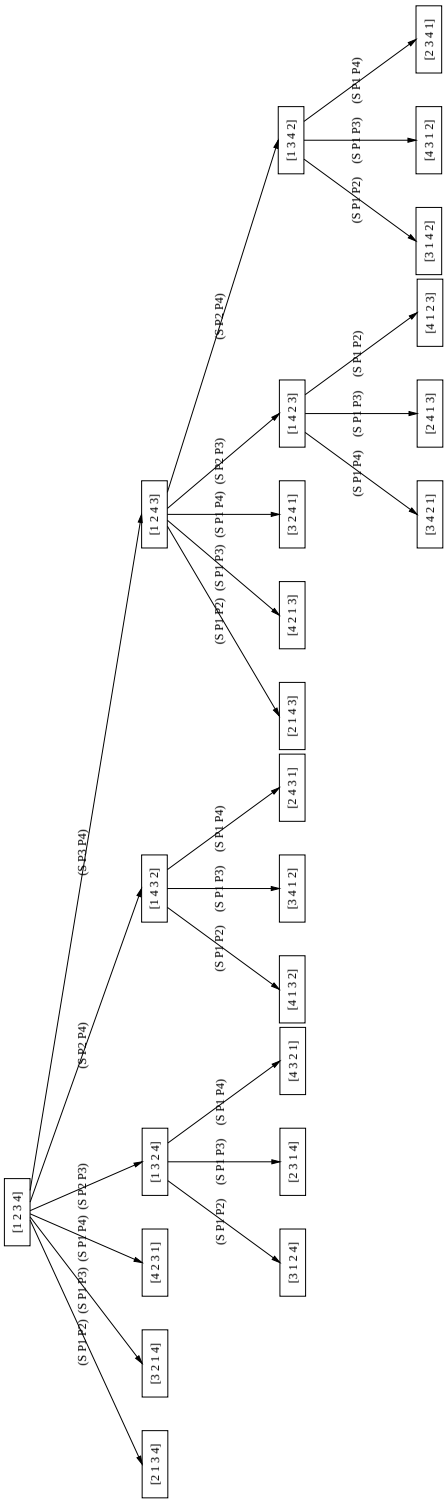



*Figure 8.15.* A Minimal Spanning Tree Extracted from the *SelSort* Plan

*Table 8.9.* Regularization of a Tree

- Input: an edge-labeled tree
- 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.

**Definition 8.7 (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 $label \leftrightharpoons label'$ if $label = label'$ or $label = id$.*

For a given minimal spanning tree it can be tried to transform it into a regular tree, using the algorithm given in table 8.9. The program fragment for tree-regularization is given in appendix A12. 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 8.7, plan-transformation proceeds, otherwise, it fails.

The regularized version of the minimal spanning tree for *selsort* is given in figure 8.16 (again with abbreviated state descriptions). 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: The actions in the bottom level of the tree all refer to swap the element on position one with some element further to the right in the list (*swap(p1, x)*). For the resulting states, the first element is already the smallest element of the list ($[1\ x\ y\ z]$). On the next level, all actions refer to swap the element on position two with some element further to the right, and so on. A slightly different method for regularization, also with the goal of plan linearization is presented in Wysotzki and Schmid (2001).

### 5.2.4 INFERRING THE SELECTOR FUNCTION

Although the *selsort* plan could be transformed successfully to a regular tree with identical subtrees, the plan is still not linear. Considering the nodes at each planning level, these nodes share a common sub-set of literals. That is, at each level, the commonalities of the states can be captured by calculating the

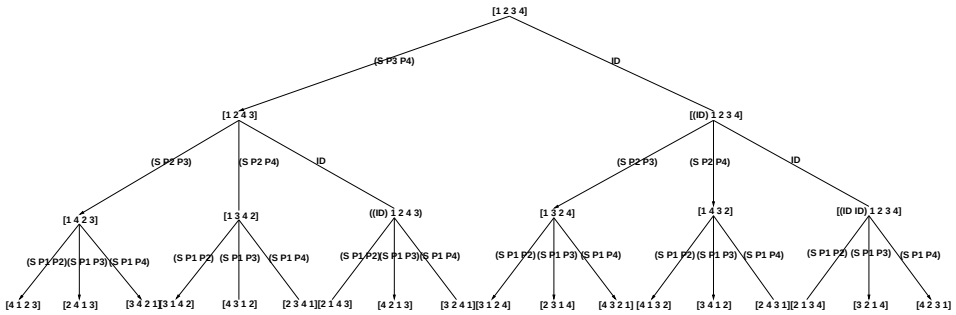



*Figure 8.16.* The Regularized Tree for *SelSort*

intersection of the state descriptions. For the branches from one node holds that their ordering is irrelevant for obtaining the action sequence for transforming a given state into the goal state. Furthermore, the names of all actions are identical and at each level, the first argument of *swap* is identical. Putting this information together, we can represent the levels of the regularized tree as:

- ((isc p1 1) (isc p2 2) (isc p3 3)) ← (swap **p3** [p4])
- ((isc p1 1) (isc p2 2)) ← (swap **p2** [p3, p4])
- ((isc p1 1)) ← (swap **p1** [p2, p3, p4]).

From the instantiations of the second argument we can construct a hypothetical 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. That is, when applying *swap*, the first position is given and the second position must be obtained somehow. A modified *swap*-function which deals with selecting the "right" position to swap must provide the following characteristics:

- Because the overall structure of the plan indicates a *list* problem, the goal predicates must be processed in a fixed order which can be derived from the sequence of actions in the plan. For a given *list* of goal predicates, deal with the first of these predicates.
  For example, the goal predicates can be *(isc p1 1) (isc p2 2) (isc p3 3)*. The first element to be considered is *(isc p1 1)*.
- A generalized *swap** operator works on the current goal predicate and the current situation $s$.
  For example, *swap** realizes *(isc p1 1)* in situation *s = (isc p1 4) (isc p2 1) (isc p3 2) (isc p4 3)*.
- The current goal is splitted into the current position and the element for this position.
  For example, *(pos (isc p1 1)) = p1* and *(key (isc p1 1)) = 1*.
- The element currently at position *(pos (isc x y))* must be swapped with the element which should be at this position, that is *(key (isc x y))*.
  For *s = (isc p1 4) (isc p2 1) (isc p3 2) (isc p4 3)* results that position $p1$ and position $p2$ are swapped.

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) )))
```

[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.

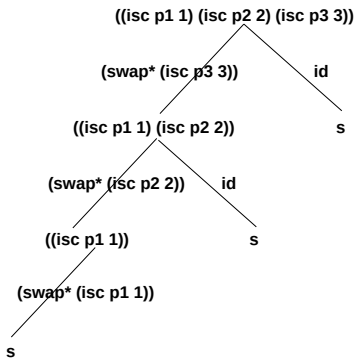



*Figure 8.17.* Introduction of a "Semantic" Selector Function in the Regularized Tree

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. 8.17). 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. Note further, that only the first argument of *swap** is given, because the second argument $s$ is given by the sub-tree following the action. This is the same way to represent plans as terms as used in the sections before (for *unstack* and *rocket*).

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

[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)`.

```
        (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 8.18.

## 5.3 CONCLUDING REMARKS ON LIST PROBLEMS

List sorting is an example for a domain which involves not only structural but also semantic knowledge: While for structural problems, the elements of the input-data can be replaced by variables with arbitrary interpretation, this is not true for semantic problems. For example, reversing a list *[x, y, z]* works in the same way, regardless whether this list is *[1, 2, 3]* or *[1, 1, 1]*. In semantic problems, such as sorting, operators comparing elements ($x < y$ or $x = y$) are necessary. A prototypical example for a problem involving semantic is the *member* function which returns true, if an element $x$ is contained in a list $l$ and false otherwise. Here an equality test must be performed. A detailed discussion of the reasons why the *member* function cannot be inferred using standard techniques of inductive logic programming is given by Le Blanc (1994).

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 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. Second, it does not rely on the ordering of natural numbers, but generates the desired sequence of elements, explicitely given as input.

We demonstrated plan transformation of 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 one to two in the regularized tree). An alternative approach to plan transformation, involving knowledge about numbers, is described for a plan for

```
; 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 8.18.* LISP-Program for *SelSort*

three-element lists in Wysotzki and Schmid (2001). 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 must to involve at least three objects to identify the recursive parameter and its substitution as described in chapter 7.

Because plan construction 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.

# 6 PLANS OVER COMPLEX DATA TYPES

## 6.1 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, *fibonacci* is typically implemented as *tree*-recursion (see tab. 8.10), 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 (Field and 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 (see chap. 7) of all functions in table 8.10 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 non-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

*Table 8.10.* 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))))) |
| $\mu$-Recursion | |
| (ack x y) == | (if (= 0 x) (1+ y) (if (= 0 y) (ack (1- x) 1) (ack (1- x) (ack x (1- y))))) |

programming (Field and Harrison, 1988). In Wysotzki and Schmid (2001) linearizability is discussed in detail.

There are two well-known planning domains, for which the underlying data type is more complex than *sets* or *lists*: *Tower of Hanoi* (sect. 1.4.4 in chap. 3) and building a *Tower* of alphabetically sorted blocks in the blocks-world domain (sect. 1.4.5 in chap. 3). The *tower* 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.

## 6.2 THE 'TOWER' DOMAIN

### 6.2.1 A PLAN FOR THREE BLOCKS

The specification of the three-block *tower* problem was introduced in chapter 2 and is described in section 1.4.5 in chapter 3. The *unstack*/*clearblock* 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 is represented with a conditioned effect for taking care of the case that a block $x$ is moved from the table to another block $y$ and for the case that $x$ is moved from another block $z$.

For the 3-block problem, the universal plan is a unique minimal spanning tree (see fig. 3.10). Note, that for a given goal to build a tower with {*on(A, B), on(B, C)*}, the state {*on(C, B), on (B, A)*} – that is a tower where the blocks are sorted in reverse order to the goal – only two actions are needed to reach the goal state: base of the tower $C$ is put on the table, $B$ can immediately put on $C$ without putting it on the table first. A program for realizing *tower* is only optimal, if these short-cuts are performed.

There are two possible strategies for learning a control program for *tower* which are discussed in reinforcement learning under the labels *incremental elemental-to-composite learning* and *simultaneous composite learning* (Sun and Sessions, 1999): In the first case, the system is first trained with a simple task, for example clearing an arbitrary block by unstacking all blocks lying on top of it. The learned policy is stored, for example as *CLEAR*-program and extends the set of available basic functions. When learning the policy for a more complex problem, such as *tower*, the already available knowledge (*CLEAR*) can be used. In the second case, the system immediately learns to solve the complex problem and the decomposition must be performed autonomously. The application of both strategies to the *tower* problem is demonstrated in Wysotzki and Schmid (2001). In the work reported here, we focus on the second strategy.

### 6.2.2 ELEMENTAL TO COMPOSITE LEARNING

Let us assume, that the control knowledge for clearing an arbitrary block is already available as a *CLEARBLOCK* function:

*CLEARBLOCK(x, s) = if(clear(x, s), s, puttable(topof(x), CLEARBLOCK(topof(x), s))).*

This function immediately returns the current state $s$, if *clear(x)* already holds in $s$, otherwise it is tried to put the block lying on top of $x$ on the table.

Now we want to learn a program for solving the more complex *tower* problem. In Wysotzki and Schmid (2001) we describe an approach based on the assumption of linearizability of the planning problem (see sect. 3.4.2 in chap. 2): It is presupposed that sub-goals can be achieved *immediately before* the corresponding top-level goal is achieved, For example, to reach a state where block $A$ is lying on block $B$, the action *put(A, B)* can be applied; but this action is applicable only if both $A$ and $B$ are clear. That is, to realize goal *on(A, B)* the sub-goals *clear(A)* and *clear(B)* must hold in the current situation. Allowing the use of the already learned recursive *CLEARBLOCK* function and using linearization, the complex plan for solving the *tower* problem for three blocks, can be collapsed to:

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

The *CLEARBLOCK* program does apply as many *puttable* actions as necessary. If a block is already clear, the state is returned unchanged. The recursive program generalizing over this plan is given in appendix C6.

For some domains, it is possible to identify independent sets of predicate by analyzing the operator specifications, for example using the TIM-analysis (see sect. 5.1 in chap. 2) for the planner STAN (Long and Fox, 1999).

While the approach of Wysotzki and Schmid (2001) is based on analyzing the goals and sub-goals contained in a plan, the strategy reported in this chapter is based on data type inference for uniform sub-plans (see sect. 2.1). If you look at the plan given in figure 3.10, the two upper levels contain only arcs labelled with *put* actions and all levels below contain only arcs labelled with *puttable* actions. Therefore, as demonstrated for *rocket* above, the plan is in a first step decomposed and it becomes impossible to infer a program where *put* and *puttable* actions are interwoven.

### 6.2.3 SIMULTANEOUS COMPOSITE LEARNING

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:

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

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 above, we have to generate a semantic selector function which is not dependent of the parent-node, but of the *current* state.[13]

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 $(A\ B\ C)$ we always must 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. The selector function is semantic, because it is relevant *which* block must be cleared. 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!

For generating optimal transformation sequences, we must cover the cases where a block can be put onto another block immediately, without putting it on the table first. For the three-block plan, there is only one such case and 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

[13]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!

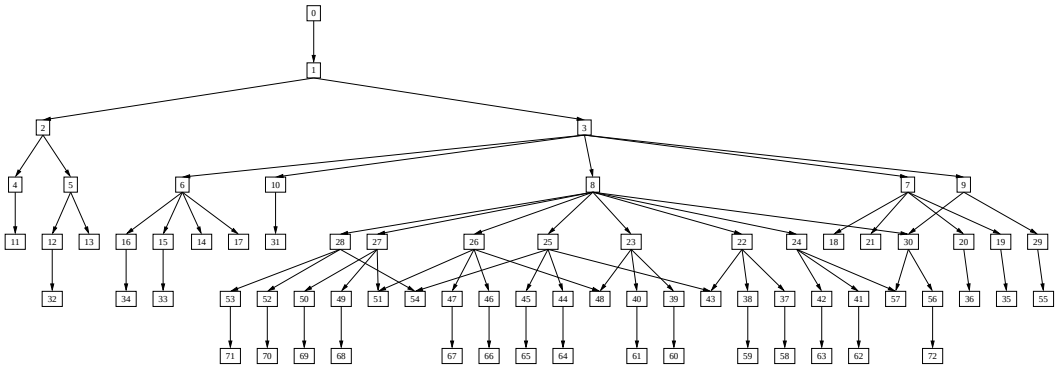



*Figure 8.19.* Abstract Form of the Universal Plan for the Four-Block *Tower*

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 8.19 and a summary for the action sequences for all 33 leaf nodes in table 8.11.

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 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 two *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)*! Again, the order is derived from the number of actions needed to transform a state into the goal state and a tower *(D C A B)* can be transformed into the goal using two *puttable* actions and three *put* actions while for *(D A B C)* three *puttable* actions and three *put* actions are needed (see tab. 8.11). 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.

*Table 8.11.* 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) | |

A program which covers all conditions for generating optimal plans is given in appendix C6.

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.

*Table 8.12.* 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)))
```

### 6.2.4 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 2.3. 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" (Knuth, 1992). The corresponding formula is $exp(x/(1-x))$.[14]

The *tower* problem is related to generating the power-set of a list with mutually different elements (see tab. 8.12). But there is also a difference between the two domains: For *powerset* each element of the set is a set again, that is, for example $\{\{a\}, \{b, c\}, \{b\}\}$ is equal to $\{\{a\}, \{c, b\}, \{b\}\}$. In contrast, for *tower*, the elements of the sets are lists. For example $\{$*(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 generated from `powerset` by changing `:test 'setequal` to `:test 'equal` in `ins-el`. The *tower* domain is the *inverse* problem to *set of lists*: For sets of *lists* a single list is decomposed in all possible partial lists. For *tower* each state corresponds to a set of partial lists and the goal state is the set containing a single list with all elements in a fixed order. 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 and putting it in front of another list `(cons (car l1) l2)`. A program generating a list of sorted numbers is given in appendix C6.

[14] 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 8.13.* 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 $(\neg(on_g = on_s) \wedge (\forall on_s.(\forall on_s^{-1}.clear_s)))$

### 6.2.5 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 (sect. 3.2 in chap. 6). 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 "evolution space" of programs. Programs generated by this approach are given in figure 6.4. These programs correspond 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 (see sect. 5.2 in chap. 2. 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 8.13. 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, that is, a learning approach where each new example is used to modify the current hypothesis if this example is not covered in the correct way. Instead, 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 C6 and from the (*still non-optimal!*) control rules in table 8.13, 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).

## 6.3 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. Tower of Hanoi can be seen as a modified *tower* problem: In contrast to *tower*, where blocks can be put on arbitrary positions on a table, in *Tower of Hanoi* the positions of discs are restricted to some (typically three) positions of pegs. This results in a more regular structure of the DAG than for the *tower* problem and therefore, we hope that if we come up with a plan transformation strategy for *tower*, the *Tower of Hanoi* domain is covered by this strategy as well.

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 and Hayes, 1976; Clément and Richard, 1997). I want to point out, that there are *solitaire* ("patience") games, which are isomorphic to *hanoi*.[15] One of these solitaire-games (freecell) was included in the AIPS-2000 planning competition.

The domain specification for *hanoi* is given in table 3.8. The resulting plan is a unique minimal spanning tree, which is already regular (see fig. 3.9). 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. 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

[15] 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 8.14.* 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)))
```

the minimal number of moves for an *arbitrary* starting state – that is, one of the nodes of the universal plan.[16]

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 8.14). In general, *hanoi* is $\mu$-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 8.14 is as tree-recursion.

We are interested in *learning* a control strategy starting with an *arbitrary initial state* (see program in table 8.15).

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

*Table 8.15.* 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 9

# CONCLUSIONS AND FURTHER RESEARCH

## 1 COMBINING PLANNING AND PROGRAM SYNTHESIS

We demonstrated that planning can be combined with inductive program synthesis by first generating a universal plan for a problem with a small number of objects, then transforming this plan into a finite program term, and folding this term into a recursive program scheme. While planning and folding can be performed by powerful, domain-independent algorithms, plan transformation is knowledge dependent. In part I, we presented the domain-indpendent universal planner DPlan. In this part, we presented an approach to folding finite program terms based on pattern-matching which is more powerful than other published approaches.

Our approach to plan transformation, presented in the previous chapter, is based on inference of the data type underlying the planning domain. Typically, such knowledge is pre-defined in program synthesis, for example, as domain-axiom – as in the deductive approach of Manna and Waldinger (1975) – or as inherent restriction of the input domain – as in the inductive approach of Summers (1977). We go beyond these approaches, providing a method to infer the data type from the structure of the plan where inference is based on a set of predefined abstract types. Furthermore, we presented first ideas for dealing with problems relying on semantic knowledge. In program synthesis, typcially, only structural list problems (such as *reverse*) are considered. Planning problems which can be solved by using structural knowledge only are for example *clearblock* and *rocket*. In *clearblock*, relations between objects (*on(x, y)*) are independent of attributes of these objects (as their size). In *rocket*, relations between objects are irrelevant. In contrast, sorting lists depends on a semantic relation between objects, that is, whether one number is greater

than another. If a universal plan has parallel branches representing the same operation but for different objects, we assume that a semantic selector function must be introduced. The selector is constructed by identifying discriminating literals between the underlying problem states (see sect. 5 in chap. 8). To sum up, with our current approach we can deal with structural problems in a fully automatic way from planning over plan transformation to folding. Dealing with problems relying on semantic information is a topic for further research.

Currently we are not exploiting the full power of the folder when generalizing over plans: In plan transformation, it is tried to come up with a single, linear structure as input to the folder although the folder allows to infer *sets* of recursive equations with arbitrarily complex recursive structures. In future research we plan to investigate possibilities for submitting complex, non-linear plans directly to the folder.

A drawback of using planning as basis for constructing finite programs is that number problems, such as *factorial* or *fibonacci*, cannot be dealt with in a natural way. For such problems, our folder must rely on input traces provided by a user. Using a graphical user interface to support the user in constructing such traces is discussed by Schrödl and Edelkamp (1999).

# 2 ACQUISITION OF PROBLEM SOLVING STRATEGIES

The most important building-stones for the flexible and adaptive nature of human cognition are powerful mechanisms of learning.[1] On the low-level end of learning mechanisms is stimulus-response learning, mostly modelled with artificial neural nets or reinforcement learning. On the high-level end are different principles of induction, that is, generalizing rules from examples. While the majority of work in machine learning focusses on induction of concepts (classification learning, see sect. 3.1.1 in chap. 6), our work focusses on inductive learning of cognitive skills from problem solving. While concepts are mostly characterized as *declarative* knowledge (know what) which can be verbalized and is accessible for reasoning processes, skills are described as highly automated *procedural* knowledge (know how) (Anderson, 1983).

## 2.1 LEARNING IN PROBLEM SOLVING AND PLANNING

Problem solving is generally realized as heuristic search in a problem space. In cognitive science most work is in the area of goal driven production systems (see sect. 4.5.1 in chap. 2). In AI, different planning techniques are investigated

[1] This section is a short version of the previous publications Schmid and Wysotzki (1996) and Schmid and Wysotzki (2000c).

(see sect. 4.2 in chap. 2). In both frameworks, the definition of problem operators together with conditions for their application – that is,, production rules – is central. Matching, selection, and application of operators is performed by an interpreter (control strategy): the preconditions of all operators defined for a given domain are matched against the current data (problem state), one of the matching operators is selected and applied on the current state. The process terminates if the goal is fulfilled or if no operator is applicable.

Skill acquisition is usually modelled as composition of predefined primitive operators as result of their co-occurrence during problem solving, that is, learning by doing. This is true in cognitive modelling (knowledge compilation in ACT Anderson and Lebière, 1998), (operator chunking in Soar Rosenbloom and Newell, 1986) as well as in AI planning (macro learning, see sect. 5.2 in chap. 2). Acquisition of such "linear" macros can result in a reduction of search, because now composite operators can be applied instead of primitive ones. In cognitive science, operator-composition is viewed as mechanism responsible for acquisition of automatisms and the main explanation for speed-up effects of learning (Anderson et al., 1989).

In contrast, in AI planning, learning of domain specific control knowledge, that is, learning of problem solving *strategies*, are investigated as an additional mechanism, as discussed in section 5.2 in chapter 2. One possibility to model acquisition of control knowledge is learning of "cyclic" macros (Shell and Carbonell, 1989; Shavlik, 1990). Learning a problem solving strategy ideally eliminates search completely because the complete sub-goal structure of a problem domain is known. For example, a macro for a one-way transportation problem as *rocket* represents the strategy that *all* objects must be loaded before the rocket moves to its destination (see sect. 1.4.2 in chap. 2 and sect. 4 in chap. 8). There is empirical evidence, for example in the Tower of Hanoi domain, that people can acquire such kind of knowledge (Anzai and Simon, 1979).

## 2.2 THREE LEVELS OF LEARNING

We propose, that our system, as given in figure 1.2 provides a general framework for modeling the acquisition of strategies from problem solving experience: Starting-point is a problem specification, given as primitive operators, their application conditions, and a problem solving goal. Using DPlan, the problem is explored, that is, operator sequences for transforming some initial states into a state fulfilling the goals are generated. This experience is integrated into a finite program, corresponding roughly to a set of operator chunks, as discussed above using plan transformation. Subsequently, this experience is generalized to a recursive program scheme (RPS) using a folding technique. That is, the system infers a domain specific control strategy which simultaneously represents the (goal) structure of the current domain. Alternatively, – as

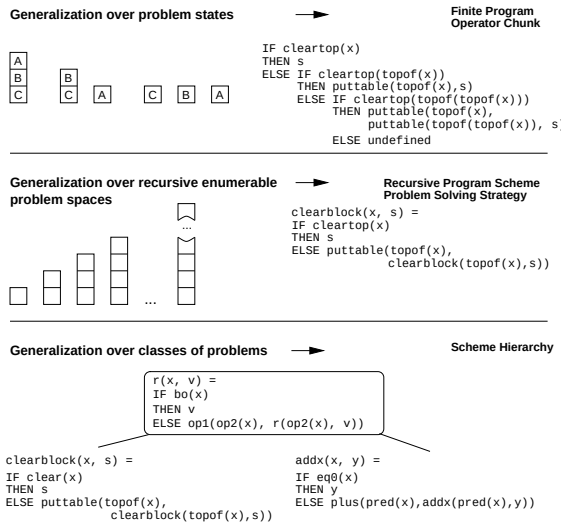



*Figure 9.1.* Three Levels of Generalization

we will discuss in part III – after initial exploration a similar problem might be retrieved from memory. That is, the system recognizes that the new problem can be solved with the same strategy as an already known problem. In this case, a further generalized scheme, representing the abstract strategy for solving both problems is learned.

All three steps of learning are illustrated in figure 9.1 for the simple *clearblock* example which was discussed in chapters 3, 7, and 8: To reach the goal that block $C$ has a clear top, all blocks lying above $C$ have to be put on the table. This can be done by applying the operator $puttable(x)$ if block $x$ has a clear top. The finite program represents the experience with three initial states as a nested conditional expression. The most important aspect of this finite program, which makes it different from the cognitive science approaches to operator chunking, is, that objects are not referred to directly by their name but with help of a selector function (*topof*). Selector functions are inferred from the structure of the universal plan (as described in chap. 8). This process in a way captures the evolution of perceptual chunks from problem solving experience (Koedinger and Anderson, 1990): For the *clearblock* example, the introduction of *topof(x)* represents the knowledge that the relevant part of the problem is the block lying on top of the currently focussed block. For more complex domains, as Tower of Hanoi, the data type represents "partial solutions" (such as how many discs are already in the correct position or looking for the largest free disc).

In the second step, this primitive behavioral program is generalized over recursive enumerable problem spaces: a strategy for clearing a block in $n$-block problems is extrapolated and interpreted as a recursive program scheme. Induction of generalized structures from examples is a fundamental characteristic of human intelligence as for example proposed by Chomsky as "language acquisition device" (Chomsky, 1959) or by Holland, Holyoak, Nisbett, and Thagard (1986). This ability to extract general rules from some initial experience is captured in the presented technique of folding of finite programs by detecting regularities.

The representation of problem schemes by recursive program schemes differ from the representation formats proposed in cognitive psychology (Rumelhart and Norman, 1981). But RPSs are capturing exactly the characteristics which are attributed to cognitive schemes, namely that schemes represent procedural knowledge ("knowledge how") which the system can interrogate to produce "knowledge that", that is, knowledge about the structure of a problem. Thereby problem schemes are suitable for modeling analogical reasoning: The acquired scheme represents not only the solution strategy for unstacking towers of arbitrary height but for all structural identical problems. Experience with a blocks-world problem can for example be used to solve a numerical problem which has the same structure by re-interpreting the meaning of the symbols of an RPS. After solving some problems with similar structures, more general schemes evolve and problem solving can be guided by *abstract* schemes.

III

# SCHEMA ABSTRACTION: BUILDING A HIERARCHY OF PROBLEM SCHEMES

Chapter 10

# ANALOGICAL REASONING AND GENERALIZATION

*"I wish you'd solve the case, Miss Marple, like you did the time Miss Wetherby's gill of picked shrimps disappeared. And all because it reminded you of something quite different about a sack of coals." "You're laughing, my dear," said Miss Marple, "but after all, that is a very sound way of arriving at the truth. It's really what people call intuition and make such a fuss about. Intuition is like reading a word without having to spell it out. A child can't do that because it has had so little experience. But a grown-up person knows the word because they've seen it often before. You catch my meaning, Vicar?" "Yes," I said slowly, "I think I do. You mean that if a thing reminds you of something else – well, it's probably the same kind of thing."*

—Agatha Christie, *The Murder at the Vicarage*, 1930

Analogical inference is a special case of inductive inference where knowledge is transferred from a known base domain to a new target domain. Analogy is a prominent research topic in cognitive science: In philosophy, it is discussed as source of creative thinking; in linguistics, similarity-creating metaphors are studied as a special case of analogical reasoning; in psychology, analogical reasoning and problem solving are researched in innumerous experiments and there exist several process models of analogical problem solving and learning; in artificial intelligence, besides some general approaches to analogy, programming by analogy and case-based reasoning are investigated.

In the following (sect. 1), we will first give an overview of central concepts and mechanisms of the field. Afterwards (sect. 2), we will introduce analogical reasoning for domains with different levels of complexity – from proportional analogies to analogical problem solving and planning. Then we will discuss programming by analogy as a special case of problem solving by analogy (sect. 3). Finally (sect. 4), we will give some pointers to literature.

*Table 10.1.* Kinds of Predicates Mapped in Different Types of Domain Comparison (Gentner, 1983, Tab. 1, extended)

| | No. of Attributes mapped to target | No. of Relations mapped to target | Example |
|---|---|---|---|
| Mere Appearance | Many | Few | A sunflower is like the sun. |
| Literal Similarity | Many | Many | The K5 solar system is like our solar system. |
| Analogy | Few | Many | The atom is like our solar system. |
| Abstraction | Few | Many | The atom is a central force system. |
| Metaphor | Many | Few | Her smile is like the sun. |
| | Few | Many | King Lois XIV was like the sun. |

# 1 ANALOGICAL AND CASE-BASED REASONING

## 1.1 CHARACTERISTICS OF ANALOGY

Typically for AI and psychological models of analogical reasoning is that domains or problems are considered as structured objects, such as terms or relational structures (semantic nets). Structured objects are often represented as graphs with basic objects as nodes and relations between objects as arcs. Relations are often distinguished in object attributes (unary relations), relations between objects (first order $n$-ary relations), and higher oder relations.

Based on this representational assumption, Gentner (1983) introduced a characterization of analogy and contrasted analogy with other modes of transfer between two domains (see tab. 10.1): In contrast to mere appearance and literal similarity, analogy is based on mapping the relational structure of domains rather than object attributes. For the famous Rutherford analogy (see fig. 10.4) – "The atom is like our solar system" – it is relevant that there is a central object (sun/nucleus) and objects *revolving around* this object (planets/electrons), but is irrelevant how much these objects weight or what their temperature is. The same is true for abstraction, but in contrast to analogy, the objects of the base domain (central force system) are generalized concepts rather than concrete instances. Metaphors can be found to be either a form of similarity-based transfer (comparable to mere appearance) or a form of analogy where a similarity is created between two previously unconnected domains (Indurkhya, 1992).[1] In contrast to analogy, case-based reasoning often relies on a simple mapping of domain attributes (Kolodner, 1993).

[1] Famous are the metaphors of Philip Marlowe. Just to give you one: "The purring voice was now as false as an usherette's eyelashes and as slippery as a watermelon seed." Raymond Chandler, *The Big Sleep*, 1939.

Below we will give a hierarchy of analogical reasoning from simple propositional analogies where only one relation is mapped to complex analogies between (planning or programming) problems. In chapter 11 we will introduce a graph representation for water-jug problems in detail.

There is often made a distinction between within- and between-domain analogies (Vosniadou and Ortony, 1989). For example, using the solution of one programming problem (*factorial*) to construct a solution to a new problem (*sum*) is considered as within-domain (Anderson and Thompson, 1989) while transferring knowledge about the solar system to explain the structure of an atom is considered as between-domain. Because analogy is typically described as *structure* mapping (see below), we think that this classification is an artificial one: When source and target domains are represented as relational structures and these structures are mapped by a *syntactical* pattern matching algorithm, the content of these structures is irrelevant. In case-based reasoning, base and target are usually from the same domain.

## 1.2 SUB-PROCESSES OF ANALOGICAL REASONING

Analogical reasoning is typically described by the following, possibly interacting, sub-processes:

**Retrieval:** For a given new target domain/problem a "suitable", "similar" base domain/problem is retrieved (from memory).

**Mapping:** The base and target structures are mapped.

**Transfer:** The target is enriched with information from the base.

**Generalization:** A structure generalizing over base and target is induced.

Typically, it is assumed that retrieval is based on superficial similarity, that is, a base problem is selected which shares a high number of attributes with the new problem. Superficial similarity and structural similarity must not necessarily correspond and it was shown in numerous studies that human problem solvers have difficulties in finding an adequate base problem (Novick, 1988; Ross, 1989).

Most cognitive science models of analogical reasoning focus on modeling the mapping process (Falkenhainer, Forbus, and Gentner, 1989; Hummel and Holyoak, 1997). The general assumption is that objects of the base domain are mapped to objects of the target domain in a structurally consistent way (see fig. 10.1). The approaches differ with respect to the constraints on mapping, allowing only first order or also higher order mapping, and restricting mapping to isomorphism, homomorphism or weaker relations (Holland et al., 1986).

Gentner (1983) proposed the following constraints for mapping a base domain to a target domain:

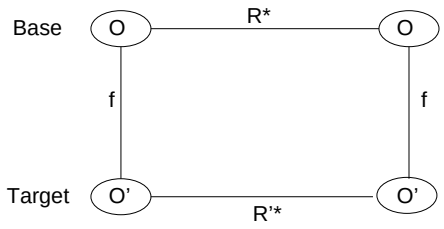



*Figure 10.1.* Mapping of Base and Target Domain

**First Order Mapping:** An object in the base can be mapped on a different object in the target, but base relations must be mapped on identical relations in the target.

**Isomorphism:** A set of base objects must be mapped one-to-one and structurally consistent to a set of target objects. That is, if $r(o_1, o_2)$ holds for the base then $r(f(o_1), f(o_2))$ must hold for the target, where $f(o)$ is the mapping function.

**Systematicity:** Mapping of large parts of the relational structure is preferred over mapping of small isolated parts, that is, relations with greater arity are preferred.

Relying on these constraints, mapping corresponds to the problem of finding the largest common sub-graph of two graphs (Schädler and Wysotzki, 1999). If the base is mapped to the target, the parts of the base graph which are connected with the common sub-graph are transferred to the target where base objects are translated to target objects in accordance with mapping. This kind of transfer is also called inference, because relations known in the base are assumed to also hold in the target. If the isomorphism constraint is relaxed, transfer additionally can involve modifications of the base structure. This kind of transfer is also called adaptation (see chap. 11).

After successful analogical transfer, the common structure of base and target can be generalized to a more general scheme (Novick and Holyoak, 1991). For example, the structure of the solar system and the Rutherford atom model can be generalized to the more general concept of a central force system. To our knowledge, none of the process models in cognitive science models generalization learning (see chap. 12).

In case-based reasoning, mostly only retrieval is addressed and the retrieved case is presented to the system user as a source of information which he might transfer to a current problem.

## 1.3 TRANSFORMATIONAL VERSUS DERIVATIONAL ANALOGY

The subprocesses described above characterize so called *transformational* analogy. An alternative approach for analogical *problem solving*, called *derivational* analogy, was proposed by Carbonell (1986). He argues that, from a computational point of view, transformational analogy is often inefficient and can result in suboptimal solutions. Instead of calculating a base/target mapping and solving the target by transfer of the base structure, it might be more efficient to *reconstruct* the solution *process*; that is, use a remembered problem solving episode as *guideline* for solving the new problem. A problem solving episode consists of the reasoning traces (derivations) of past solution processes, including the explored subgoal structure and used methods. Derivational analogy can be characterized by the following subprocesses: (1) Retrieving a suitable problem solving episode, (2) applying the retrieved derivation to the current situation by "replaying" the problem solving episode, checking for each step if the derivation is still applicable in the new problem solving context. Derivational analogy is for example used within the AI planning system *Prodigy* (Veloso and Carbonell, 1993). An empirical comparison of transformational and derivational analogy was conducted by Schmid and Carbonell (1999).

## 1.4 QUANTITIVE AND QUALITATIVE SIMILARITY

As mentioned above, retrieval is typically based on similarity of attributes. For comparison of base and target, all kinds of similarity measures defined on attribute vectors can be used. An overview over measures of similarity and distance is typically given in textbooks on cluster analysis, such as Eckes and Roßbach (1980). In psychology, non-symmetrical measures are discussed, for example, the contrast-model of Tversky (1977).

In analogy research, focus is on measures for *structural* similarity. A variety of measures are based on the size of the greatest common sub-graph of two structures. Such a measure for un-labeled graphs was for example proposed by Zelinka (1975): $d(G,H) = |N| - |N_U|$, where $\mid N \mid$ is the number of nodes of the larger graph and $\mid N_U \mid$ is the number of nodes in the greatest common sub-graph of $G$ and $H$. A measure considering the number of nodes and arcs in the graphs is introduced in chapter 11.

Another approach to structural similarity is to consider the number of transformations which are necessary for making two graphs identical (Bunke and Messmer, 1994). A measure of transformation distance for trees was for example proposed by Lu (1979): $d(T_1,T_2) = w_s \cdot s + w_d \cdot d + w_i \cdot i$, where $s$ represents the number of substitutions (renamings of node-labels), $d$ the number of deletions of nodes, and $i$ the number of insertion of nodes. Parameters $w$ give operation specific weights. To guarantee that the measure is a

metric, it must hold that $w_s < w_d, w_i$ and $w_d = w_i$ (proof in Mercy, 1998). Transformational similarity is also the basis for structural information theory (Leeuwenberg, 1971).

All measures discussed so far are *quantitative*, that is, a numerical value is calculated which represents the similarity between base and target. Usually, a threshold is defined and a domain or problem is considered as a candidate for analogical reasoning, if its similarity to the target problem lies above this threshold. A *qualitative* approach to structural similarity was proposed by Plaza (1995). Plaza's domain of application is the retrieval of typed feature terms (e. g., records of employees), which can be hierarchically organized, for example the field "name" consists of a further feature term with the fields "first name" and "surname". For a given (target) feature term, the most similar (base) feature term from a data base is identified by the following method: Each pair of the fixed target and a term from the data base is anti-unified. The resulting anti-instances are ordered with respect to their subsumption relation and the most specific anti-instance is returned. Anti-unification was introduced in section 3.3 in chapter 6 under the name of least general generalization. An anti-unification approach for programming by analogy is presented in chapter 12.

# 2 MAPPING SIMPLE RELATIONS OR COMPLEX STRUCTURES

## 2.1 PROPORTIONAL ANALOGIES

The most simple form of analogical reasoning are proportional analogies of the form

*A:B :: C:?* *"A is to B as C to ?"*.

Expression $A$ to $B$ is the base domain and expression $C$ to ? is the target domain. The relation existing between $A$ and $B$ must be identified and applied to the target domain. Such problems are typically used in intelligence tests, and algorithms for solving proportional analogies were introduced early in AI research (Evans, 1968; Winston, 1980). In intelligence tests, items are often semantic categories as "Rose is to Flower as Herring to ?". In this example, the relevant relation is subordinate/superordinate and the superordinate concept to "Herring" must be retrieved (from semantic memory). Because the solution of such analogies is knowledge dependent, often letter strings (Hofstadter and The Fluid Analogies Research Group, 1995; Burns, 1996) or simple geometric figures (Evans, 1968; Winston, 1980; O'Hara, 1992) are used as alternative.

An example for an analogy which can be solved with Evans Analogy-program (Evans, 1968) is given in figure 10.2. In a first step, each figure is decomposed into simple geometric objects (such as a rectangle and a triangle). Because decomposition is ambiguous for overlapping objects, Evans

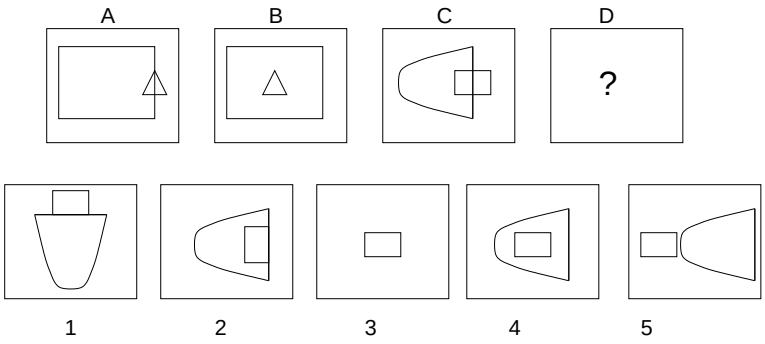



*Figure 10.2.* Example for a Geometric-Analogy Problem (Evans, 1968, p. 333)

used Gestalt-laws and context-information for decomposition. For the pairs of figures $(A, B)$, $(A, C)$, $(B, C)$, $(C, 1)$, ..., $(C, 5)$ relations between objects and between spatial positions are constructed. These relations are used for constructing transformation rules $A \rightarrow B$. Transformation includes object mapping, deletion and insertion. The rules are abstracted such that they can be applied to figure $C$, resulting in a new figure which hopefully corresponds to one of the given alternatives for $D$.

An algorithm which can use geometric analogy problems using context-dependent re-descriptions was proposed by O'Hara (1992). An example is given in figure 10.3. All figures must be composed of lines. An algebra is used to represent/construct figures. Operations are translation, rotation, reflection, scaling of objects and glueing of pairs of objects.

In a first step, an initial representation of figure $A$ is constructed. Then, a representation of $B$ and a transformation $t$ are constructed such that $B = t(A)$. A representation of $C$ and an isomorphic mapping $\mu$ are constructed such that $C = \mu(A)$. Mapping $\mu$ is extended to $\mu'$, preserving isomorphism, and $D = \mu'(B)$ is constructed. If no mapping $C = \mu(A)$ can be found, the algorithm backtracks and constructs a different representation for $B$ (re-description).

Letter string domains are easier to model as geometric domains and different approaches to algebraic representation (Leeuwenberg, 1971; Indurkhya, 1992) have been proposed. The Copycat program of Hofstadter and The Fluid Analogies Research Group (1995) solves letter string analogies of the form $abc : abd :: kji :?$.

## 2.2 CAUSAL ANALOGIES

In proportional analogies, a domain is represented by two structured objects $A$ and $B$ with a single or a small set of relations $t(A) = B$ which are relevant for analogical transfer. More complex domains are explanatory structures, for

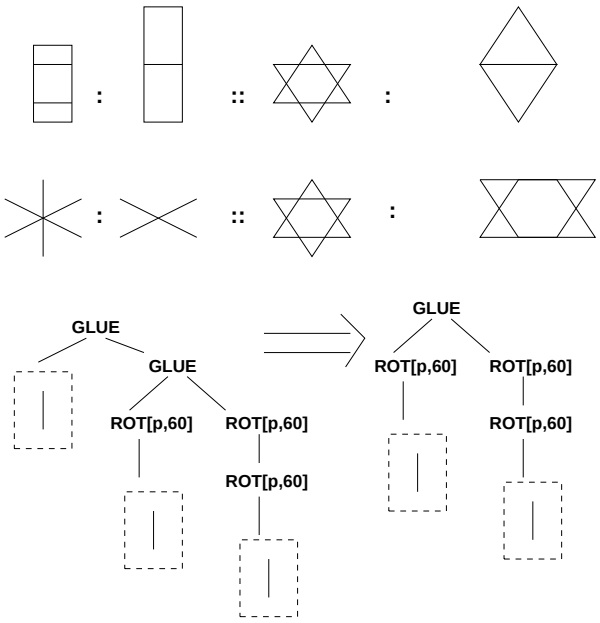



*Figure 10.3.* Context Dependent Descriptions in Proportional Analogy (O'Hara, 1992)

example for physical phenomena. In Rutherford analogy, introduced above, knowledge about the solar system is used to infer the *cause* why electrons are rotating around the nucleus of an atom (see fig. 10.4).

Systems realizing mapping and inference for such domains are for example the structure mapping engine (*SME*, Falkenhainer et al., 1989) or *LISA* (Hummel and Holyoak, 1997). The *SME* is based on the constraints for structure mapping proposed by Gentner (1983), which were introduced above. Note, that also the inferred higher-order relation is labeled "cause", mapping and inference are purely syntactical, as described for proportional analogies.

## 2.3 PROBLEM SOLVING AND PLANNING BY ANALOGY

Many people have pointed out that analogy is an important mechanism in problem solving. For example, Polya (1957) presented numerous examples for how analogical reasoning can help students to learn proving mathematical theorems. Psychological studies on analogical problem solving address mostly solving of mathematical problems (Novick, 1988; Reed, Ackinclose, and Voss, 1990) or program construction (Anderson and Thompson, 1989; Weber, 1996). Typically, students are presented with worked out *examples* which they can use

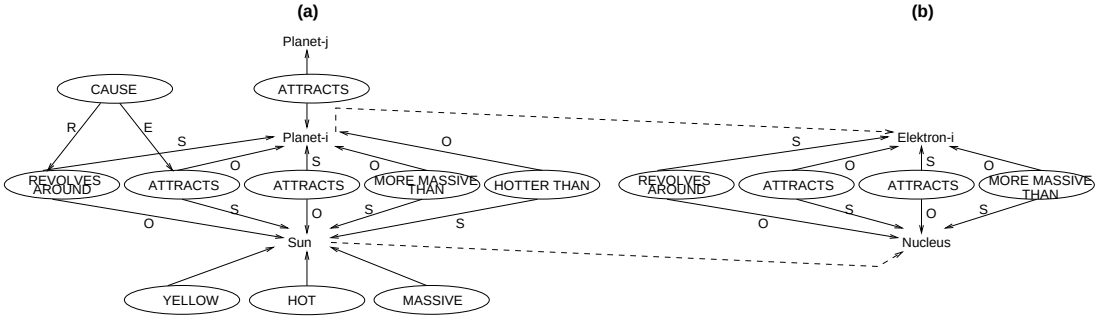



*Figure 10.4.* The Rutherford Analogy (Gentner, 1983)

to solve a new (target) problem. The standard example for programming is using *factorial* as base problem to construct *sum*:

*fac(x) = if(eq0(x), 1, *(x, fac(pred(x))))*
*sum(x) = if(eq0(x), 0, +(x, sum(pred(x))))*.

Anderson and Thompson (1989) propose an analogy model where programming problems and solutions are represented as schemes with function slots representing the intended operation of the program and form slots representing the syntactical realization. For a target problem only the function slot is filled and the form slot is inferred from the given base problem.

Examples for simple word algebra problems used in experiments by Reed et al. (1990) is given in table 10.2. Analogical problem solving means to identify the relations between concepts in a given problem with the equation for calculating the solution. For a new (target) problem, the concepts in the new text have to be mapped with the concepts in the base problem and then the equation can be transferred. Reed et al. (1990) could show that problem solving success is higher if a base problem which *includes* the target problem is used (e. g., the third problem in tab. 10.2 as base and the second as target) but that most subjects did not select the inclusive problems as most helpful if they could choose themselves. In chapter 11 we report experiments with different variants of inclusive problems in the water jug domain.

An AI system which addresses problem solving by analogy is *Prodigy* (Veloso and Carbonell, 1993). Here, a derivational analogy mechanism is used (see above). *Prodigy-Analogy* was, for example, applied for the *rocket* domain (see sect. 1.4.2 in chap. 3): A planning episode for solving a *rocket* problem is stored (e. g., transporting two objects) and indexed with the initial and goal states. For a new problem (e. g., transporting four objects) the old episode can be retrieved by matching the current initial and goal states with the

*Table 10.2.* Word Algebra Problems (Reed et al., 1990)

| |
|---|
| A group of people paid \$238 to purchase tickets to a play. How many people were in the group if the tickets cost \$14 each?<br>\$14 = \$238/n |
| A group of people paid \$306 to purchase theater tickets. When 7 more people joined the group, the total cost was \$425. How many people were in the original group if all tickets had the same price?<br>\$306/n = \$425/(n+7) |
| A group of people paid \$70 to watch a basketball game. When 8 more people joined the group the total cost was \$ 20. How many people were in the original group if the larger group received a 20% discount?<br>$0.8 \cdot (\$70/n) = \$120/(n+8)$ |

indices of stored episodes and the retrieved episode is replayed. While Veloso and Carbonell (1993) could demonstrate efficiency gains of reuse over planning from the scratch for some example domains, Nebel and Koehler (1995) provided a theoretical analysis that in the worst case reuse is not more efficient than planning from the scratch. The first bottleneck is retrieval of a suitable case and the second is modification of old solutions.

## 3 PROGRAMMING BY ANALOGY

Program construction can be seen as a special case of problem solving. In part II we have discussed automatic program synthesis as an area of research which tries to come up with mechanisms to automatize or support – at least routine parts of – program development. Programming by analogy is a further such mechanism, which is discussed since the beginning of automatic programming research. For example, Manna and Waldinger (1975) claim that retention of previously constructed programs is a powerful way to acquire and store knowledge.

Dershowitz (1986) presented an approach to construct imperative programs by analogical reasoning. A base problem – e. g., calculating the division of two real numbers with some tolerance – is given by its specification and solution. The solution is annotated with additional statements (such as *assert*, *achieve*, *purpose*) which makes it possible to relate specification and program. For a new problem – e. g., calculating the cube-root of a number with some tolerance – only the specification is given. By mapping the specifications (see fig. 10.5), the statements in the base program are transformed into statements of the to be constructed target program. Some additional inference methods are used to

Real-Division:
ASSERT $0 <= c < d, e > 0$
ACHIEVE $|c/d - q| < e$
VARYING $q$

Cube-Root:
ASSERT $a >= 0, e > 0$
ACHIEVE $|a^{(1/3)} - r| < e$
VARYING $r$

Mapping: $q \Rightarrow r, c/d \Rightarrow a^{(1/3)}$
Transformations:
$q \rightarrow r$,
$u/v \rightarrow u^{(1/3)}$ (*replace each division operator by a cube-root operator*),
$c \rightarrow a$

*Figure 10.5.* Base and Target Specification (Dershowitz, 1986)

generate a program which is correct with respect to the specification from the initial program which was constructed by analogy.

Both programs are based on the more general strategy of binary search. As a last step of programming by analogy, Dershowitz proposes to construct a scheme by abstracting over the programs. For the example above, the operators for division and cube-root are generalized to $\gamma(u, v)$, that is, a second-order variable is introduced. New binary search problems can then be solved by instantiating the scheme. If program schemes are acquired, deductive approaches to program synthesis can be applied, for example, the stepwise refinement approach used in the KIDS system (see sect. 2.2.3 in chap. 6).

Crucial for analogical programming is to detect relations between pairs of programs. The theoretical foundation for constructing mappings are program morphisms (Burton, 1992; Smith, 1993). A technique to perform such mappings is anti-unification. A completely implemented system for programming by analogy, based on second-order anti-unification and generalization morphisms was presented by Hasker (1995). For a given specification and program, the system user provides a program *derivation* (c. f., derivational analogy), that is, steps for transforming the specification into the program. Second-order anti-unification is used to detect analogies between pairs of specifications which are represented as combinator terms. First, he introduces anti-unification for monadic combinator terms, that is, allowing only unary terms. Then, he introduces anti-unification for product types. Monadic combinator terms are not expressive enough to represent programs while combinator terms for cartesian product types – allowing that sub-terms are deleted – are too general and allow infinitely many minimal generalizations. Therefore, Hasker introduces relevant combinator terms as a subset of combinators for cartesian product types which allow introducing pairs of terms but do not allow to ignore sub-terms. For this class of combinator terms there still exist, possibly large, sets of minimal generalization. Therefore, Hasker introduces some

heuristics and allows for user interaction to guide construction of a "useful" generalization.

Our own approach to programming with analogy is still at its beginning. We consider how folding of finite program terms into recursive programs (see chap. 7) can be alternatively realized by analogy or abstraction. That is, mapping is performed on pairs of finite programs, where for one program the recursive generalization is known and for the other not (see chap. 12). In contrast to most approaches to analogical reasoning, retrieval is not based on attribute similarity but on structural similarity. We use anti-unification for retrieval (Plaza, 1995) as well as for mapping and transfer (Hasker, 1995). Calculating the anti-instance of two terms results in their maximally specific generalization. Our current approach to anti-unification is much more restricted than the approach of Hasker (1995). Our restricted approach guarantees that the minimal generalization of two terms is unique. But for retrieval, based on identifying the maximal specific anti-instance in a subsumption hierarchy, as proposed by Plaza (1995, see above), typically sets of candidates for analogical transfer are returned. Here, we must provide additional information to select a useful base. One possibility is, to consider the size and type of structural overlap between terms. We conducted psychological experiments to identify some general criteria of structural base/target relations which allow successful transfer for human problem solvers (see chap. 11). If a finite program (associated with its recursive generalization) is selected as base, the term substitutions calculated while constructing the anti-instance of this program with the target can be applied to the recursive base program and thereby the recursive generalization for the target is obtained (see chap. 12).

We believe, that human problem solvers prefer abstract schemes over concrete base problems to guide solving novel problems and use concrete problems as guidance only, if they are inexperienced in a domain. Therefore, in our work we focus on abstraction rather than analogy. Our approach to anti-unification works likewise for concrete program terms and terms containing object and function variables, that is schemes. Anti-unifying a new finite program term with the unfolding of some program scheme results in a further generalization. Consequently, a hierarchy of abstract schemes develops over problem solving experience.

Our notion of representing programs as elements of some term algebra instead of some given programming language (see chap. 7) allows us to address analogy and abstraction in the domain of programming as well as in more general domains of problem solving (such us solving blocks-world problems or other domains considered in planning) within the same approach. That is, as discussed in chapter 9, we address the acquisition of problem schemes which represent problem solving strategies (or recursive control rules) from experience.

## 4 POINTERS TO LITERATURE

Analogy and case-based reasoning is an extensively researched domain. A bibliography for both areas can be found at `http://www.ai-cbr.org`. Classical books on case-based reasoning are Riesbeck and Schank (1989) and Kolodner (1993). A good source for current research are the proceedings of the international conference on case-based reasoning (ICCBR). Cognitive models of analogy are presented, for example, at the annual conference of the Cognitive Science Society (CogSci) and in the journal *Cognitive Science*. A discussion of metaphors and analogy is presented by Indurkhya (1992). Holland et al. (1986) discuss induction and analogy from perspectives of philosophy, psychology, and AI. A collection of cognitive science papers on similarity and analogy was presented by Vosniadou and (Eds.) (1989). Some AI papers on analogy can be found in Michalski et al. (1986).

Chapter 11

# STRUCTURAL SIMILARITY IN ANALOGICAL TRANSFER

*"And what is science about?" "He explained that scientists formulate theories about how the physical world works, andthen test them out by experiments. As long as the experiments succeed, then the theories hold. If they fail, the scientists have to find another theory to explain the facts. He says that, with science, there's this exciting paradox, that diosillusionment needn't be defeat. It's a step forward."*

—P. D. James, *Death of an Expert Witness*, 1977

In this chapter, we present two psychological experiments to demonstrate that human problem solver do and can use not only base (source) domains which are isomorphic to target domains but also rely on non-isomorphic structures in analogical problem solving. Of course, not every non-isomorphic relation is suitable for analogical transfer. Therefore, we identified which degree of structural overlap must exist between two problems to guarantee a high probability of transfer success. This research is not only of interest in the context of theories of human problem solving. In the context of our system IPAL, criteria are needed for deciding whether it is worthwhile to try to generate a new recursive program by analogical transfer of a program (or by instantiating a program scheme) given in memory or to generate a new solution from scratch, using inductive program synthesis. In the following, we first (sect. 1) give an introduction in psychological theories to analogical problem solving and present the problem domain used in the experiments. Then we report two experiments on analogical transfer of non-isomorphical source problems (sect. 2 and sect. 3). We conclude with a discussion of the empirical results and possible areas of application (sect. 4).[1]

[1]This chapter is based on the papers Schmid, Wirth, and Polkehn (2001) and Schmid, Wirth, and Polkehn (1999).

# 1 ANALOGICAL PROBLEM SOLVING

Analogical reasoning is an often used strategy in everyday and academic problem solving. For example, if a person already has experience in planning a trip by train, he/she might transfer this knowledge to planning a trip by plane. If a student already has knowledge in solving an equation with one additive variable, he/she might transfer the solution procedure to an equation with a multiplicative variable. For analogical transfer, a previously solved problem – called source – has to be similar to the current problem – called target. While a large number of common attributes might help to *find* an analogy, source and target have to be *structurally* similar for transfer success (Holyoak and Koh, 1987). In the ideal case, source and target are structurally identical (isomorph) – but this is seldom true in real-live problem solving.

Analogical problem solving is commonly described by the following (possibly interacting) component processes (e. g., Keane, Ledgeway, and Duff, 1994): representation of the target problem, retrieval of a previously solved source problem from memory, mapping of the structures of source and target, transfer of the source solution to the target problem, and generalizing over the common structure of source and target. The empirically best explored processes are retrieval and mapping (see Hummel and Holyoak, 1997, for an overview). Retrieval of a source is assumed to be guided by overall semantic similarity (i. e., common attributes), often characterized as "superficial" in contrast to structural similarity (Gentner and Landers, 1985; Holyoak and Koh, 1987; Ross, 1989). Empirical results show that retrieval is the bottleneck in analogical reasoning and often can only be performed successfully if explicit hints about a suitable source are given (Gick and Holyoak, 1980; Gentner, Ratterman, and Forbus, 1993). Therefore, a usual procedure for studying mapping and *transfer* is to circumvent retrieval by explicitly presenting a problem as a helpful example (Novick and Holyoak, 1991). In the following, we will give a closer look at mapping and transfer.

## 1.1 MAPPING AND TRANSFER

Mapping is considered the core process in analogical reasoning. The decision whether two problems are analogous is based on identifying structural correspondences between them. Mapping is a necessary but not always sufficient condition for successful transfer (Novick and Holyoak, 1991). There are numerous empirical studies concerning the mapping process (c. f., Hummel and Holyoak, 1997) and all computational models of analogical reasoning provide an implementation of this component (Falkenhainer et al., 1989; Keane et al., 1994; Hummel and Holyoak, 1997). Mapping is typically modelled as first identifying the corresponding components of source and target and then carrying over the conceptual structure from the source to the target. For ex-

ample, in the Rutherford analogy (the atom is like the solar system), planets can be mapped to electrons and the sun to the nucleus of an atom together with relations as "revolves around" or "more mass than" (Gentner, 1983). In the structure mapping theory (Gentner, 1983) it is postulated that mapping is performed purely syntactically and that it is guided by the principle of systematicity – preferring mapping of greater portions of structure to mapping of isolated elements. Alternatively, Holyoak and colleagues postulate that mapping is constrained by semantic and pragmatic aspects of the problem (Holyoak and Thagard, 1989; Hummel and Holyoak, 1997). Mapping might be further constrained such that it results in easy adaptability (Keane, 1996). Currently, it is discussed that a target might be re-represented, if source/target mapping cannot be performed successfully (Hofstadter and The Fluid Analogies Research Group, 1995; Gentner, Brem, Ferguson, Markman, Levidow, Wolff, and Fobus, 1997).

Based on the mapping of source and target, the conceptual structure of the source can be transferred to the target. For example, the explanatory structure that the planets revolve arround the sun because the sun attracts the planets might be transferred to the domain of atoms. Transfer can be faulty or incomplete, even if mapping was performed successfully (Novick and Holyoak, 1991). Negative transfer can also result from a failure in prior sub-processes – construction of an unsuitable representation of the target, retrieval of an inappropriate source problem, or incomplete, inconsistent or inappropriate mapping of source and target (Novick, 1988). Analogical transfer might lead to the induction of a more general schema which represents an abstraction over the common structure of source and target (Gick and Holyoak, 1983). For example, when solving the Rutherford analogy, the more general concept of central force systems might be learned.

## 1.2 TRANSFER OF NON-ISOMORPHIC SOURCE PROBLEMS

Our work focusses on analogical transfer in problem solving. There is a marginal and a crucial difference between general models of analogical reasoning and models of analogical problem solving. While in general source and target might be from different domains (between-domain analogies as the Rutherford analogy), in analogical problem solving source and target typically are from the same domain (within-domain analogies, e. g., Vosniadou and Ortony, 1989). For example, people can use a previously solved algebra word problem as an example to facilitate solving a new algebra word problem (Novick and Holyoak, 1991; Reed et al., 1990), or they can use a computer program with which they are already familiar as an example to construct a new program (Anderson and Thompson, 1989). While the discrimination of between- and within-domain analogies is relevant for the question of how a

suitable source can be retrieved, it has no impact on structure mapping if this process is assumed to be performed purely syntactically.

The more crucial difference between models of analogical reasoning and of problem solving is that in analogical reasoning transfer is mostly described by inference (in so-called explanatory analogies, e. g., Gentner, 1983) vs. by adaptation (in problem solving, e. g., Keane, 1996). In the first case, (higher-order) relations given for the source are carried over to the target – as the explanation given above of why electrons revolve around the nucleus. In analogical problem solving, on the other hand, most often the complete solution procedure of the source problem is adapted to the target. Analogical transfer of a problem solution subsumes the structural "carry-over" along with possible changes (adaptation) of the solution structure and the application of the solution procedure. For example, if we are presented with the necessary operations to isolate a variable in an equation, we can solve a new equation by adapting the known solution procedure. If structures of source and target are identical (isomorphic), transfer can be described as simply replacing the source concepts by the target concepts in the source solution. For a source equation $2 \cdot x + 5 = 9$ with solution $x = \frac{(9-5)}{2}$, the target $3 \cdot x + 4 = 16$ can be solved by (1) mapping the numbers of source and target, that is $2$ is mapped to $3$, $4$ to $5$ and $9$ to $16$ and by (2) substituting the corresponding numbers in the source solution. An example for source inclusive source/target pair mapping is given below in figure 11.3.

We are especially interested in conditions for successful transfer of non-isomorphic source solutions. There are a variety of non-isomorphical source/target relations discussed in literature: First, there are different types of mapping relations: one-to-one-mappings (isomorphism), many-to-one, and one-to-many mappings (Spellman and Holyoak, 1996). Secondly, there are different types and degrees of structural overlap (see fig. 11.1): a source might be "completely contained" in the target (source inclusiveness; Reed et al., 1990), or a source might represent all concepts needed for solving the target together with some additional concepts (target exhaustiveness; Gentner, 1980). These are two special cases of structural overlap between source and target. It seems plausible to assume that if the overlap is too small, a problem is no longer helpful for solving the target. Such a problem would not be characterized as a source problem. While there are some empirical studies investigating transfer of non-isomorphic sources (Reed et al., 1990; Novick and Hmelo, 1994; Gholson, Smither, Buhrman, Duncan, and Pierce, 1996; Spellman and Holyoak, 1996), there is no systematic investigation of the structural relation between source and target which is necessary for succesful transfer. Our experimental work focusses on the impact of different types and degrees of *structural overlap* on transfer success, that is, we currently are only considering one-to-one mappings.

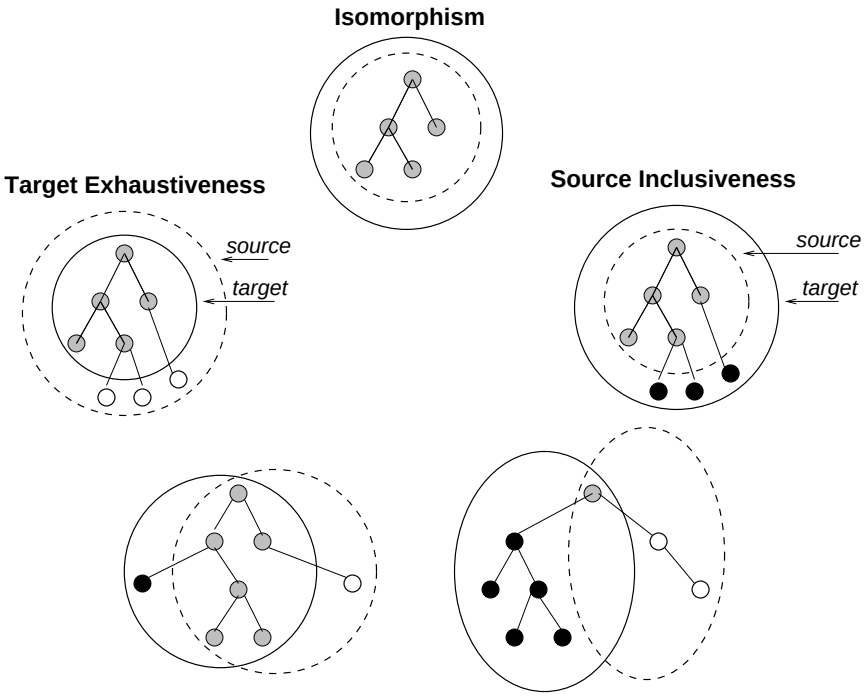



*Figure 11.1.* Types and degrees of structural overlap between source and target Problems

## 1.3 STRUCTURAL REPRESENTATION OF PROBLEMS

To determine the structural relation between source and target we have to rely on explicitly defined representations of problem structures. In cognitive models of analogical reasoning, problems are typically represented by schemas (SME, Falkenhainer et al., 1989), (IAM, Keane et al., 1994) or by semantic nets (ACME, Holyoak and Thagard, 1989), (LISA, Hummel and Holyoak, 1997). From a more abstract view, these representations correspond to graphs, where concepts are represented as nodes and relations between them as arcs. Examples for graphs are given in figure 11.1. For actual problems, nodes (and possibly arcs) are labelled. A graph representation of the solar system contains for instance a node labelled with the relation *more mass than* connected to a node *planet-1* and to a node *sun*.

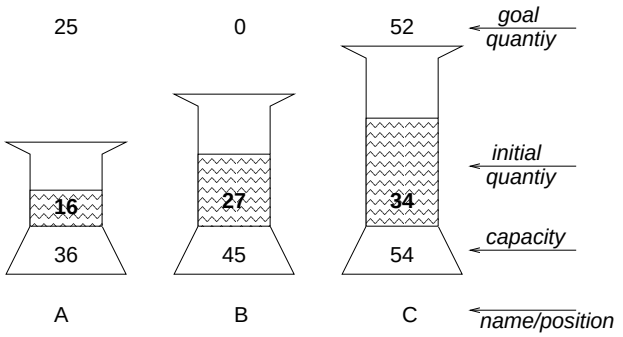
While explicit representations are often presented for explanatory analogies (Gick and Holyoak, 1980; Gentner, 1983), this is not true for problem solving. For algebra problems (Novick and Holyoak, 1991; Reed et al., 1990), the mathematical equations can be used to represent the problem structure (see fig. 11.3). In general – when investigating such problems as the Tower of Hanoi

(Clément and Richard, 1997; Simon and Hayes, 1976) – both the structure of a problem and the problem solving operators, possibly together with application conditions and constraints (Gholson et al., 1996), have to be taken into account.

In the classical transformational view of analogical problem solving (Gentner, 1983), little work has been done which addresses how to model analogical transfer of problems involving several solutions steps. In artifical intelligence, Carbonell (1986) proposed derivational analogy for multi-step problems: He models problem solving by analogy as deriving a solution by replay of an already known solution process, checking on the way whether the conditions for operator applications of the source still hold when solving the target. In the following, we nevertheless adopt the transformational approach – describing analogical problem solving by *mapping* and transfer. That is, we will assume that both the (declarative) description of the problem and procedural information are structurally represented and that a target problem is solved by adapting the source solution to the target problem based on structure mapping. We assume that problems and solutions are represented in schemas capturing declarative as well as procedural aspects as argued for example by Anderson and Thompson (1989) and Rumelhart and Norman (1981).

When specifying the representation of a problem, we have to decide on its format as well as its content (Gick and Holyoak, 1980). In general, it is not possible to determine all possible aspects associated with a problem, that is, we cannot claim complete representations. We adopt the position of Gick and Holyoak, to model at least all aspects which are relevant for successful transfer. A component of the (declarative) description of a problem is relevant, if it is necessary for generating the operation sequence which solves the problem. Furthermore, only the operation sequence which solves the problem is regarded as relevant procedural information. A successful problem solver has to focus on these relevant aspects of a problem and should ignore all other aspects. Of course, we do not assume that human problem solvers in general represent *only* relevant or *all* relevant aspects of a problem. Our goal is to systematically control variants of structural source/target relations and their impact on transfer, that is, our representational assumptions are not empirical claims but a means for task analysis. We want to construct "normatively complete" graph representations of problems to explore the impact of different analytically given structural source/target relations on empirically observable transfer success.

In the following, we will first introduce our problem solving domain – water redistribution tasks – and our problem representations. Then we will present two experiments. In the first experiment we will show that problem solvers can transfer a source solution with moderate structural similarity to the target if the problems do not vary in superficial features. In the second experiment we investigate a variety of different structural overlaps between source and target.




*Solution:* `pour(C,B), pour(B,A), pour(A,C), pour(B,A)`

*Figure 11.2.* A water redistribution problem

## 1.4 NON-ISOMORPHIC VARIANTS IN A WATER REDISTRIBUTION DOMAIN

Because we focus on transfer of declarative and procedural aspects of problems we constructed a problem type that can be classified as interpolation problems like the Tower of Hanoi problems (Simon and Hayes, 1976), the water jug problems (Atwood and Polson, 1976), or missionary-cannibal problems (Reed, Ernst, and Banerji, 1974; Gholson et al., 1996). Problem solving means to find the correct multi-step sequence of operators that transform an initial state into the goal state. Interpolation problems have well-defined initial and goal states and usually one well-defined multi-step solution. Thus, they are as suitable for systematically analyzing their structure as, for instance, mathematical problems (Reed et al., 1990) with the advantage that they are not likely to activate school-trained mathematical pre-knowledge.

We constructed a water redistribution domain that is similar to but more complex than the water jug problems described by Atwood and Polson (1976). In the initial state three (or four) jugs of different capacity are given. The jugs are initially filled with different amounts of water (initial quantities). The water has to be redistributed between the jugs in such a way that the pre-specified goal quantities are obtained. For example, given are the three jugs $A$, $B$ and $C$ with capacities $c_A = 36$, $c_B = 45$, and $c_C = 54$ (units). In the initial state quantities are $q_A = 16$, $q_B = 27$, and $q_C = 34$. To reach the goal state the values of these quantities must be transformed into $q_A = 25$, $q_B = 0$ and $q_C = 52$ by redistributing the water among the different jugs (see fig. 11.2).

The task is to determine the shortest sequence of operators that transform the initial quantities into the goal quantities. The only legal operator available is a *pour*-operator (the redistribute operator) that is restricted by the following conditions: (1) The only water to pour is the water contained by the jugs in

the initial state. (2) Water can be poured only from a non-empty 'pour out'-jug into an incompletely filled 'pour in'- jug. (3) Pouring always results in either filling the 'pour in'-jug up to its capacity with possibly leaving a rest of water in the 'pour out'-jug or emptying the 'pour out'-jug with possibly remaining free capacity in the 'pour in'-jug. (4) The amount of water that is poured out of the 'pour out'-jug is always the same amount that is filled in the 'pour in'-jug. Formally this *pour*-operator is defined in the following way:

IF not($q_X(t) = 0$) AND not($q_Y(t) = c_Y$) THEN pour(X,Y) *resulting in:*

| | |
|---|---|
| IF | $q_X(t) \leq c_Y - q_Y(t)$ |
| THEN | $q_Y(t+1) := q_Y(t) + q_X(t)$ |
| | $q_X(t+1) := q_X(t) - q_X(t)$ (i. e. 0, emptying jug X) |
| ELSE | $q_X(t+1) := q_X(t) - (c_Y - q_Y(t))$ |
| | $q_Y(t+1 := q_Y(t) + (c_Y - q_Y(t))$ (i. e., $c_Y$, filling jug Y) |

*with*
$q_X(t)$: quantity of jug $X$ at solution step *t*
$c_X$: capacity of jug $X$,
$(c_X - q_X(t))$: remaining free capacity of jug $X$.

Because we are interested in which types and degrees of structural overlap are sufficient for successful analogical transfer, we have to ensure that subjects really refer to the source for solving the target problem. That is, the problems should be complex enough to ensure that the correct solution can not be found by trial and error, and difficult enough to ensure that the abstract solution principle is not immediately inferable. Therefore, we constructed redistribution problems for which exists only a single (for two problems two) shortest operator sequence (in problem spaces with over 1000 states and more than 50 cycle-free solution paths).

To construct a structural representation of a problem we were guided by the following principles: (a) the goal quantity of each jug can be described as the initial quantity transformed by a certain (shortest) sequence of operators; (b) relevant declarative attributes (capacities, quantities and relations between these attributes) of the initial and the goal state determine the solution sequence of operators; and (c) each solution step can be described by the definition of the pour-operator given above. In terms of these principles operator applications can be re-formulated by equations where a current quantity can be expressed by adding or subtracting amounts of water. For example, using the parameters introduced above, the first operator *pour(C,B)* of the example presented in figure 11.2 transforms the quantities of jug $B$ and $C$ of the initial state $t = 0$ into quantities of state $t = 1$ in the following way: $q_B(0) = 27$, $q_C(0) = 34$, $c_B = 45$, $c_C = 54$:

BECAUSE not$(34 = 0)$ AND not$(27 = 45)$
*pour(C,B)* at $t = 0$ results in:

*Table 11.1.* Relevant information for solving the source problem

(a) Procedural

| | pour(C,B) | pour(B,A) | pour(A,C) | pour(B,A) |
|---|---|---|---|---|
| $q_A(4) =$ | $q_A(0)$ | $+(c_A - q_A(0))$ | $-c_A$ | $+[c_B - (c_A - q_A(0))]$ |
| $q_B(4) =$ | $q_B(0)$ | $+(c_B - q_B(0))$ | $-(c_A - q_A(0))$ | $-[c_B - (c_A - q_A(0))]$ |
| $q_C(4) =$ | $q_C(0)$ | $-(c_B - q_B(0))$ | | $+c_A$ |

(b) Declarative (constraints)

| | |
|---|---|
| $c_B \geq$ | $(c_A - q_A(0))$ |
| $c_A =$ | $2 \cdot (c_B - q_B(0))$ |
| $q_C(0) \geq$ | $(c_B - q_B(0))$ |
| $c_C <$ | $q_C(0) + c_A$ |

BECAUSE not(34 $\leq$ (45 - 27)):
$q_B(1)$ = 27 + (45 - 27) = 45
$q_C(1)$ = 34 - (45 - 27) = 16.

Redistribution problems define a specific goal quantity for each jug. For this reason, there have to be as many equations constructed as there are jugs involved. This group of equations represents all relevant procedural aspects of the problem, that is, it represents the sequence of operators leading to the goal state. For example, the three equations of the three jugs in figure 11.2 are presented in table 11.1.

Each goal state is expressed by the values given in the initial state. On the right side of the equality sign these given values are combined in a way that transforms the initial quantity of each jug into its goal quantity. Certain constraints between the initial values have to be satisfied so that these equations are balanced. These constraints can be analytically derived from the equations given in table 11.1 and they constitute the relevant declarative attributes of the problem. The constraints for water redistribution problems have the same function as the problem solving constraints given for example for the radiation or fortress problems (Holyoak and Koh, 1987) or the missionary-cannibal problems (Reed et al., 1974; Gholson et al., 1996).

As an example of how these contraints can be derived from the equations, you can easily see that the last pour operator of the solution (pouring a certain amount of water from $B$ into $A$) is only executable if the relation $c_B \geq (c_A - q_A(0))$ holds. As a second constraint, the goal quantity of jug $C$ could be described by the following equation $q_C(4) = q_C(0) + (c_B - q_B(0))$. But the expression $(c_B - q_B(0))$ does not represent the quantity in jug $B$. It represents the remaining free capacity of this jug which, of course, can not be poured into

another jug. Because the relation $c_A = 2 \cdot (c_B - q_B(0))$ holds, we conclude that the value of the remaining free capacity of jug $B$ has to be subtracted from the quantity of jug $C$ (only possible if $q_C(0) \geq (c_B - q_B(0))$ holds) and that the double of this value ($c_A$) has to be added to the quantity in jug $C$ by pouring the capacity of jug $A$ into jug $C$. Additionally, the relation $c_C < q_C(0) + c_A$ determines the relative order of the two pour-operators: you have to subtract $(c_B - q_B(0))$ before you can add $c_A$ to $q_C(0)$.

The equations describing the transformations for each jug and the (in-)equations describing the constraints of the problem are sufficient to represent all relevant declarative and procedural information of the problem. Thus, transforming all of them into one graphical representation leads to a normatively complete representation of the problem which can be used for a task analytical determination of the overlap between two problem structures. The equations and in-equations for all water redistribution problems used in our experiments are given in appendix C7.

## 1.5 MEASUREMENT OF STRUCTURAL OVERLAP

Structural similarity between two graphs $G$ and $H$ is usually calculated as the size of the greatest common subgraph of $G$ and $H$ in relation to the size of the greater of both graphs (Schädler and Wysotzki, 1999; Bunke and Messmer, 1994). To calculate graph distance by formula 1 we introduce directed "empty" arcs between all pairs of nodes where no arcs exist. The size of the common subgraph is expressed by the sum of common arcs $V_{GH}$ and nodes $N_{GH}$.

$$d_{(G,H)} = 1 - \frac{V_{GH} + N_{GH}}{max(V_G, V_H) + max(N_G, N_H)} \tag{11.1}$$

The graph distance can assume values between 0 and 1, indicating isomorphic relations between two graphs with $d_{(G,H)} = 0$ and no systematic relation between $G$ and $H$ with $d_{(G,H)} = 1$. For the two partial isomorphic graphs in figure 11.3 we obtain (for graph 11.3.a as $G$ and graph 11.3.b as $H$)

| | | |
|---|---|---|
| $V_G = 42$ | $V_H = 72$ | $(V_G < V_H)$ |
| $N_G = 7$ | $N_H = 9$ | $(N_G < N_H)$ |
| $V_{GH} = 30$ | $N_{GH} = 6$ | |

resulting in the difference between G and H

$$d_{(G,H)} = 1 - \frac{30 + 6}{72 + 9} = 0.57.$$

The value of $d_{(G,H)} = 0.57$ indicates that the size of the common subgraph of $G$ and $H$ is a little more than half of the size of the larger graph $H$. Of course,

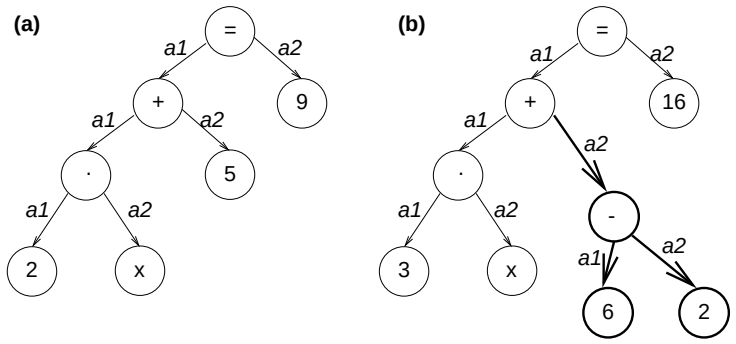



*Figure 11.3.* Graphs for the equations $2 \cdot x + 5 = 9$ (a) and $3 \cdot x + (6 - 2) = 16$ (b)

this absolute value of the distance between two problems is highly dependent on the kind of representation of their structures. Thus, for task analysis only the ordinal information of these values should be considered.

## 2 EXPERIMENT 1

Experiment 1 was designed to investigate the suitability of the water-redistribution for studying analogical transfer in problem solving, to get some initial information about transfer of isomorphic vs. non-isomorphic sources, and to check for possible interaction of superficial with structural similarity.

The problems were constructed in such a way that it is highly improbable that the correct optimal operator-sequence can be found by trial-and-error or that the general solution principle is immediately inferable. To investigate analogical *transfer*, information about mapping of source and target can be given before the target problem has to be solved (Novick and Holyoak, 1991). This can be done by pointing out the relevant properties of a problem (conceptual mapping) and by giving information about the corresponding jugs in source and target ("numerical" mapping). Additionally, information about the problem solving strategy of subjects can be obtained by analyzing log-files of subjects' problem solving behavior and by testing mapping after subjects solved the target problem.

To get an indication of the degree of structural similarity between a source and a target which is necessary for transfer success, an isomorphic source/target pair was contrasted with a partial isomorphic source/target pair with "moderately high" structural overlap. This should give us some information about the range of source/target similarities which should be investigated more closely (in experiment 2).

To control possible interactions of structural and superficial similarity, we discriminate structure preserving and structure violating variants of target prob-

lems (Holyoak and Koh, 1987): For a given source problem with three jugs (see fig. 11.2), a target problem with four jugs clearly changes the superifical similarity in contrast to a target problem consisting also of three jugs. But this additional jug might or might not result in a change of the problem structure – reflected in the sequence of *pour* operations necessary for solving the problem. In contrast, other superficial variations – like changing the sequence of jugs from small/medium/large to large/medium/small – are structure preserving, but clearly affect superficial similarity. If the introduction of an additional jug does not lead to *additional* deviations from the surface appearance of the source problem, we regard the surface as "stable". As a consequence, there are four possible source/target variations: structure preserving problems with stable or changed surface and structure violating problems with stable or changed surface.

## 2.1 METHOD

### Material

As source problem, the problem given in figure 11.2 was used. We constructed five different redistribution problems as target problems (see appendix C7):

**Problem 1:** a three jug problem solvable with four operators which is isomorphic to the source problem (condition *isomorph/no surface change*),

**Problem 2:** a three jug problem solvable with four operators which is isomorphic to the source problem, but has a surface variation by switching positions of the small jug (A) and the medium jug (B) and renaming these jugs accordingly ($A \rightarrow B$, $B \rightarrow A$) (condition *isomorph/small surface change*),

**Problem 3:** a three jug problem solvable with four operators which is isomorphic to the source problem, but has a surface variation by switching positions of all jugs ($A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$) (condition *isomorph/large surface change*),

**Problem 4:** a four jug problem solvable with five operators which has a moderately high structural overlap with the source (condition *partial isomorph/no surface change*), and

**Problem 5:** a four jug problem solvable with five operators which is isomorph to problem 4, but has a surface variation by switching positions of two jugs ($A \rightarrow B$, $B \rightarrow A$) (condition *partial isomorph/small surface change*).

Because of the exploratory nature of this first experiment, we did not introduce a complete crossing of structure and surface similarity. The main question was, whether subjects could successfully use a partial isomorph in analogical transfer.

In addition to the source and target problems, an "initial" problem which is isomorphic to the source was constructed. This problem was introduced *before* the presentation of the source problem for the following reasons: first, subjects should become familiar with interacting with the problem solving environment (the experiment was fully computer-based, see below); and second, subjects should be "primed" to use analogy as a solution strategy, by getting demonstrated how the source problem could be solved with help of the initial problem.

### Subjects

Subjects were 60 pupils (31 male and 29 female) of a gymnasium in Berlin, Germany. Their average age was 17.4 years (minimum 14 and maximum 19 years).

### Procedure

The experiment was fully computer based and conducted at the school's PC-cluster. The overall duration of an experimental session was about 45 minutes. All interactions with the program were recorded in log-files. One session consisted of the following parts:

**Instruction and Training.** First, general instructions were given, informing about the following tasks and the water-redistribution problems. Subjects were introduced to the setting of the screen-layout (graphics of the jugs) and the handling of interactions with the program (performing a *pour*-operation, un-doing an operation).

**Initial problem.** Afterwards, the subjects were asked to solve the initial problem with tutorial guidance (performed by the program). In case of correct solution the tutor asked the subject to repeat it without any error. The tutor intervened, if subject had performed four steps without success, or had started two new attempts to solve the problem, or if they needed longer than three minutes. This part was finished, if the problem was correctly solved twice without tutorial help.

**Source problem.** When introducing the source problem, first some hints about the relevant problem aspects for figuring out a shortest operator-sequence were given (by thinking about the goal quantities in terms of relations to initial quantities and maximum capacities). Afterwards, the correspondance between the three jugs of the initial problem and the three jugs of the source problems were pointed out. Now, the screen offered an additional button to retrieve the solution sequence of the initial problem. The initial solution could be retrieved as often as the subject desired. To perform an operation for solving the source problem, this window had to be closed. Tutorial

*Table 11.2.* Results of Experiment 1

| **Problem** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| structure[a] | ISO | ISO | ISO | P-ISO | P-ISO |
| surface[b] | no | small | large | no | small |
| n | 12 | 12 | 12 | 12 | 12 |
| solved | 12 | 11 | 9 | 8 | 6 |
| correct mapping | 8 | 12 | 10 | 9 | 6 |
| shortest solution | 8 | 10 | 8 | 8 | 3 |
| transfer rate | 100% | 83.3% | 80% | 88.9% | 50% |

[a] ISO = isomorph, P-ISO = partial isomorph

[b] no, small, large change of surface

guidance was identical to the initial problem, but subjects had to solve the source problem only once.

**Target problem.** Every subject randomly received one of the five target problems. Again, mapping hints (relevant problem aspects, correspondance between jugs of the source and the target problem) were given. The source solution could be retrieved without limit, but, again, the subjects could only proceed to solve the target problem after this window was closed. Thereby, the number and time of reference to the source problem could be obtained in log-files. The subjects had a maximum of 10 minutes to solve the target problem.

**Mapping Control.** Mapping success was controlled by a short test where subjects had to give relations between the jugs of the source and target problem.

**Questionnaire.** Finally, mathematical skills (last mark in mathematics, subjective rating of mathematical knowledge, interest in mathematics) and personal data (age and gender) were obtained.

## 2.2 RESULTS AND DISCUSSION

Overall, there was a monotonic decrease in problem solving success over the five target problems (see tab. 11.2, lines *n* and *solved*). To make sure, that problem solving success was determined by successful *transfer of the source* and not by some other problem solving strategy, only subjects which gave a correct mapping were considered and a solution was rated as transfer success if the generated solution sequence was the (unique) shortest solution (see tab. 11.2, lines *correct mapping* and *shortest solution*). The variable "transfer success" was calculated as percentage of subjects whith correct mapping which generated the shortest solution (see tab. 11.2, line *transfer rate*).

Log-file analysis showed, that none of the subjects who performed correct mapping, retrieved the source solution while solving the target problem. It is highly improbable that the correct shortest solution was found randomly or that subjects could infer the general solution principle when solving the initial and source problem. As a consequence, we have to assume that these subjects solved the target by analogical transfer of the memorized (four-step) source solution.

Transfer success decreased nearly monotonicly over the five conditions. Exceptions were problems 3 (*isomorph/high surface change*) and 4 (*partial isomorph/no surface change*). The high percentage of transfer success for problem 4 indicates clearly, that subjects can succesfully transfer a non-isomorphic source problem. Even for problem 5 (*partial isomorph/surface change*) transfer success was 50%.

There is no overall significant difference between the five experimental conditions (exact $5 \times 2$ polynomial test[2]: $P = 0.21$). To control interactions between structural and superficial similarity, different contrasts were calculated which we discuss in the following.

### 2.2.1 ISOMORPH STRUCTURE/CHANGE IN SURFACE

There is no significant impact of the variation of superficial features between conditions 1, 2 and 3 (exact binomial-tests: 1 vs. 2 with $P = 0.225$; 2 vs. 3 with $P = 0.558$; and 1 vs.3 with $P = 0.168$). This finding is in contrast to Reed et al. (1990). Reed and colleagues showed that subjects' rating of the suitability of a problem for solving a given target is highly influenced by superficial attributes. However, these ratings were obtained *before* subjects had to solve the target problem. This indicates, that superficial similarity has a high impact on retrieval of a suitable problem but not on transfer success, as was also shown by Holyoak and Koh (1987) when contrasting structure-preserving vs. structure-violating differences.

### 2.2.2 CHANGE IN STRUCTURE/STABLE SURFACE

Changes in superficial attributes between conditions 1 and 4 (isomorph vs. partial isomorph, both with no surface change) respectively 2 and 5 (isomorph vs. partial isomorph, both with small surface change) can be regarded as stable, because the additional jug (in condition 4 vs. 1 and condition 5 vs. 2) influences only structural attributes. That is, by contrasting these conditions we measure the influence of structural similarity on transfer success. There is no significant difference between condition 1 and 4 (exact binomial-tests: 1 vs.

[2]For this test, the result has to be tested against the number of cell-value distributions corresponding with the given row and column values. The procedure for obtaining this number was implemented by Knut Polkehn.

4 with $P = 0.394$). But there is a significant difference between conditions 2 and 5 (exact binomial-tests: 2 vs. 5 with $P = 0.039$): A partial isomorph can be useful for analogical transfer if it shares superficial attributes with the target, but, transfer difficulty is high if source and target vary in superficial attributes – even if the mapping is explicitly given!

### 2.2.3 CHANGE IN STRUCTURE/CHANGE IN SURFACE

The variation of superficial attributes between conditions 4 and 5 has a significant impact (exact binomial-tests: $P = 0.039$). As shown for the contrast of conditions 2 and 5, if problems are not isomorphic, superficial attributes gain importance. Of course, this finding is restricted to the special type of source/target pairs and variation of superficial attributes we investigated – that is, to cases where the target is "larger" than the source and where jugs are always named as $A$, $B$, $C$ ($D$), but the names can be associated with jugs of different sizes. In this special case, the intuitive constraint of mapping jugs with identical names and positions has to be overcome and kept active during transfer.

To summarize, this first explorative experiment shows, that water redistribution problems are suitable for investigating analogical transfer – most subjects could solve the target problems, but solution success is sensitive to variations of source/target similarity. As a consequence of the interaction found between superficial and structural similarity, in the following, superficial source/target similarity will be kept high for all target problems and we will investigate only target problems varying in their structural similarity to the source (i. e., with stable surface). Finally, the high solution success for the partial isomorph of "moderately high" structural similarity (condition 4) indicates, that we can investigate source/target pairs with a smaller degree of structural overlap.

## 3 EXPERIMENT 2

In the second experiment we investigated a finer variation of different types and degrees of structural overlap. We focused on two hypotheses about the influence of structural similarity on transfer:

(1) We have been interested in the possibly different effects of different types of structural overlap on transfer – that is target exhaustiveness versus source inclusiveness of problems (c. f., fig. 11.1). If one considers a problem structure as consisting of *only relevant* declarative and procedural information, different types of structural relations result in differences in the amount of both common relevant declarative and common relevant procedural information. Changing the amount of common declarative information requires ignoring declarative source information in the case of target exhaustiveness and additionally identifying declarative target information in the case of source inclusiveness. Changing the amount of common procedural information means

changing the length of the solution (i. e., the minimal number of *pour*-operators necessary to solve the problem).

Thus, compared to the source solution target exhaustiveness results in a shorter target solution while the target solution is longer for source inclusiveness. Assuming that ignoring information is easier than additionally identifying information (Schmid, Mercy, and Wysotzki, 1998) and assuming that a shorter target solution is easier to find than a longer one, we expect that successful transfer should be more probable for target exhaustive than for source inclusive problems. In line with this assumption, Reed et al. (1974) reported increasing transfer frequencies for target exhaustive relations, if subjects were informed about the correspondences between source and target (see also Reed et al., 1990).

(2) While source inclusiveness and target exhaustiveness are special types of structural overlap we have also been interested in the overall impact of the degree of structural overlap on transfer. We wanted to determine the minimum size of the common substructure of source and target problem that makes the source useful for analogically solving the target. Or in other words, we wanted to measure the degree of the distance between source and target structures up to which the source solution is transferable to the target problem.

## 3.1 METHOD

### Material

As initial and source problem we used the same problems as in experiment 1. As target problems we constructed following five different redistribution problems with constant superficial attributes (see appendix C7):

**Problem 1:** a three jug problem solvable with three operators whose structure was completely contained in the structure of the source problem (condition *target exhaustiveness*),

**Problem 2:** a three jug problem solvable with five operators whose structure contained completely the structure of the source problem (condition *source inclusiveness*),

**Problem 3:** the partial isomorph problem used before (condition 4 in experiment 1) – a four jug problem solvable with five operators whose structure completely contains the structure of the source problem; this problem shares a smaller structural overlap with the source than problem 2 (condition *"high" structural overlap*), and

**Problem 4 and 5:** more four jug problems solvable with five operators that have decreasing structural overlap with the source; the structures of source and target share a common substructure, but both structures have additional

aspects (conditions *"medium" structural overlap* and *"low" structural overlap*)

For all problem structures distances to the source structure were calculated using formula 1 (see appendix C7). Because of the intrinsic constraints of the water redistribution domain, it was not possible to obtain equi-distance between problems. Nevertheless, the problems we constructed served as good candidates for testing our hypotheses.

To investigate the effect of the *type* of structural source/target relation, the distances of problem 1 (target exhaustive) and problem 2 (source inclusive) to the source have been kept as low as possible and as similar as possible: $d_{S1} = 0.16$ and $d_{S2} = 0.17$. As discussed above, it can be expected, that target exhaustiveness leads to a higher probability of transfer success than source inclusiveness.

Although problem 3 is a source inclusive problem, we used it as an anchor problem for varying the *degree* of structural overlap. Target problem 4 differed moderately from target problem 3 in its distance value ($d_{S3} = 0.37$ vs $d_{S4} = 0.55$) while target problem 5 differed from target problem 4 only slightly ($d_{S4} = 0.55$ vs. $d_{S5} = 0.59$). Thus, one could expect strong differences in transfer rates between condition 3 and 4 and nearly the same transfer rates for conditions 4 and 5. We name problems 3, 4, and 5 as "high", "medium" and "low" overlap in accordance to the ranking of their distances to the source problem.

### Subjects

Subjects were 70 pupils (18 male and 52 female) of a gymnasium in Berlin, Germany. Their average age as 16.3 years (minimum 16 and maximum 17 years). The data of 2 subjects was not logged due to technical problems. Thus, 68 logfiles were available for data analysis.

### Procedure

The procedure was the same as in experiment 1. Each subject had to solve one initial problem and one isomorphic source problem first and was then presented one of the five target problems.

## 3.2 RESULTS AND DISCUSSION

49 subjects mapped the jugs from the source to the target correctly. Thus 19 subjects had to be excluded from analysis. Table 11.3 shows the frequencies of subjects who performed the correct mapping between source and target and generated the shortest solution sequenc, i. e., solved the target problem analogically (c. f., experiment 1).

*Table 11.3.* Results of Experiment 2

| **Problem** structure | **1** target exhaustive | **2** source inclusive | **3** "high" overlap | **4** "medium" overlap | **5** "low" overlap |
|---|---|---|---|---|---|
| n | 11 | 10 | 16 | 15 | 16 |
| correct mapping | 7 | 8 | 13 | 9 | 12 |
| shortest solution | 6 | 7 | 10 | 5 | 1 |
| transfer rate | 86% | 88% | 77% | 56% | 8% |

### 3.2.1 TYPE OF STRUCTURAL RELATION

There is no difference in solving frequencies between condition 1 and 2 (exact binomial test, $P = 0.607$). That is, there is no indication of an effect of the type of structural source/target relation on transfer success. In contrast to the findings of Reed et al. (1974) and Reed et al. (1990), it seems, that the degree of structural overlap has a much larger influence than the type of structural relation between source and target. Furthermore, looking only at the procedural aspect of our problems, we could not find an impact of the length of the required operator-sequence (three steps for problem 1 vs. 5 steps for problem 2) on solution success.

A possible explanation might be that the type of structural relation has no effect, if problems are very similar to the source. It is clearly a topic for further investigation, to check whether target exhaustive problems become superior to source inclusive problems with increasing source/target distances.

A general superiority of degree over type of overlap could be explained by assuming mapping as a symmetrical instead of an asymmetrical (source to target) process. Hummel and Holyoak (1997) argue that during retrieval the target representation "drives" the process. In contrast, during mapping the role of the "driver" can switch from target to source and vice versa. During transfer the source structure again takes control of the process. Thus, an interaction between these processes must lead to decreasing differences between effects of source inclusiveness and effects of target exhaustiveness on analogical transfer.

### 3.2.2 DEGREE OF STRUCTURAL OVERLAP

Each problem of conditions 3 to 5 has been solvable with at least five operators. That means, there was one additional operator needed compared to the source solution. Results for conditions 3 to 5 show a significant difference between the effects of different degrees of structural source/target overlap on solution frequency (exact $3 \times 2$ test, $P = 0.002$). Comparing each single frequency against each other indicates that the crucial difference between

structural distances is between conditions 4 and 5 (exact binomial test, conditions 3 and 4: $P = 0.1$; conditions 3 and 5: $P < 0.001$; conditions 4 and 5: $P = 0.0001$).

This finding is surprising taking into account that the difference of structural distance between conditions 3 and 4 is much larger than between condition 4 and 5 ($d_{S3} = 0.37$, $d_{S4} = 0.55$, $d_{S5} = 0.59$). A possible explanation is, that with problem 5 we have reached the margin of the range of structural overlap where a problem can be helpful for solving a target problem. A conjecture worth further investigation is, that a problem can be considered as a suitable source if it shares at least fifty percent of its structure with the target! An alternative hypothesis is, that not the *relative* but rather the *absolute* size of structural overlap determines transfer success – that is, that a source is no longer helpful to solve the target, if the number of nodes contained in the common sub-structure gets smaller than some fixed lower limit.

## 4 GENERAL DISCUSSION

In our studies we investigated only a small selection of analytically possible source/target relations. We did not investigate many-to-one versus one-to-many mappings (Spellman and Holyoak, 1996), and we only looked at target exhaustiveness versus source inclusiveness for problems with a large common structure. We plan to investigate these variations in further studies. For source/target relations with a varying degree of structural overlap we were able to show that a problem is suitable as source even if it shares only about half of its structure with the target. A first explanation for this finding which goes along with models of transformational analogy is, that subjects first construct a partial solution guided by the solution for the structurally identical part of the solution, and than use this partial solution as a constraint for finding the missing solution steps by some problem solving strategy, such as means-end-analysis (Newell, Shaw, and Simon, 1958), or by internal analogy (Hickman and Larkin, 1990).

Internal analogy describes a strategy where a previously ascertained solution for a part of a problem guides the construction of a solution for another part of the same problem. For the problem domain we investigated, internal analogy gives no plausible explanation: The constraints used to figure out the solution steps for the overlapping part of the target problem are not the same as those used for the non-overlapping part – therefore internal analogy cannot be applied. A second explanation might be, that subjects try to re-represent the target problem in such a way that it becomes isomorphic to the source (Hofstadter and The Fluid Analogies Research Group, 1995; Gentner et al., 1997). Again, this explanation seems to be inplausible for our domain: Because the number of jugs and given initial, goal, and maximum quantities determine the solution

steps completely, re-representation (for example looking at two different jugs as one jug) cannot be helpful for finding a solution.

The results of the present study give some new insights about the nature of structural similarity underlying transfer success in analogous problem solving. While it is agreed upon that application of analogies is mostly influenced by structural and not by superficial similarity (Reed et al., 1974; Gentner, 1983; Holyoak and Koh, 1987; Reed et al., 1990; Novick and Holyoak, 1991), there are only few studies that have investigated which type and what degree of structural relationship between a source and a target problem is necessary for transfer success.

Holyoak and Koh (1987) used variants of the radiation problem (Duncker, 1945) to show that structural differences have an impact on transfer. They varied structural similarity by constructing problems with different solution constraints. In studies using variants of the missionaires-cannibales problem structural similarity was varied in the same way (Reed et al., 1974; Gholson et al., 1996). In the area of mathematical problem solving, typically the complexity of the solution procedure is varied (Reed et al., 1990; Reed and Bolstad, 1991). While in all of these studies non-isomorphic source/target pairs are investigated, in none of them the type and degree of structural similarity was controlled. Thus, the question of which structural characteristics make a source a suitable candidate for analogical transfer remained unanswered.

Investigating structural source/target relations is of practical interest for several reasons: (1) In an educational context (cf. tutoring systems) the provided examples have to be carefully balanced to allow for generalization (learning). Presenting only isomorphs restricts learning to small problem classes, while too large a degree of structural dissimilarity can result in failure of transfer and thereby obstructs learning (Pirolli and Anderson, 1985). (2) A plausible cognitive model of analogical problem solving (Falkenhainer et al., 1989; Hummel and Holyoak, 1997) should generate correct transfer only for such source/target relations where human subjects perform successfully. (3) Computer systems which employ analogical or case-based reasoning techniques (Carbonell, 1986; Schmid and Wysotzki, 1998) should refrain from analogical transfer when there is a high probability of constructing faulty solutions. Thus, situations can be avoided in which system users have to check – and possibly debug – generated solutions. Here information about conditions for successful transfer in human analogical problem solving can provide guidelines for implementing criteria when the strategy of analogical reasoning should be rejected in favour of other problem solving strategies.

Chapter 12

# PROGRAMMING BY ANALOGY

In part II we discussed inductive program synthesis as an approach to learning recursive program schemes. Alternatively, in this chapter, we investigate how a recursive program scheme can be generalized from two, structurally similar programs and how a new recursive program can be constructed by adapting an already known scheme. Generalization can be seen as the last step of programming or problem solving by analogy. Adaptation of a scheme to a new problem can be seen as abstraction rather than analogy because an abstract scheme is applied to a new problem – in contrast to mapping two concrete problems. In the following, we first (sect. 1) give a motivation for programming by analogy and abstraction. Afterwards (sect. 2), we introduce a restricted approach to second-order anti-unification. Then we present first results on retrieval (sect. 3), introducing subsumption as a qualitative approach to program similarity. In section 4, generalization of recursive program schemes as anti-instances of pairs of programs is described. Finally (sect. 5), some preliminary ideas for adaptation are presented.[1]

## 1 PROGRAM REUSE AND PROGRAM SCHEMES

It is an old claim in software engineering, that programmers should write less code but reuse code developed in previous efforts (Lowry and Duran, 1989). In the context of programming, reuse can be characterized as transforming an old program into a new program by replacing expressions (Cheatham, 1984; Burton, 1992). But reuse is often problematic on the level of concrete programs because the relation between code fragments and the function they perform is not always obvious (Cheatham, 1984). There are two approaches to overcome

[1]This chapter is based on the previous publication Schmid, Sinha, and Wysotzki (2001).

this problem: *(1)* performing transformations on the program specifications rather than on the concrete programs (Dershowitz, 1986), and *(2)* providing abstract schemes which are stepwise refined by "vertical" program transformation (Smith, 1985). The second approach proved to be quite successful and is used in the semi-automatic program synthesis system KIDS (Smith, 1990, see sect. 2.2.3 in chap. 6).

In the KIDS system, program schemes, such as divide-and-conquer, local, and global search, are predefined by the system author and selection of an appropriate scheme for a new programming problem is performed by the system user. From a software-engineering perspective, it is prudent to exclude these knowledge-based aspects of program synthesis from automatization. Nevertheless, we consider automatic retrieval of an appropriate program scheme and automatic construction of such program schemes from experience as an interesting research questions.

While the KIDS system is based on deductive (transformational) program synthesis, the context of our own work is *inductive* program synthesis: A programming problem is specified by some input/output examples and an universal plan is constructed which represents the transformation of each possible input state of the initially finite domain into the desired output (see part I). This plan is transformed into a finite program tree which is folded into a recursive function (see part II). It might be interesting to investigate reuse on the level of programming problems – that is, providing a hypothetical recursive program for a new set of input/output examples omitting planning. But currently we are investigating how the folding step can be replaced by analogical transfer or abstraction and how abstract schemes can be generalized from concrete programs.

Our overall approach is illustrated in figure 1.1: For a given finite program (T) representing some initial experience with a problem that program scheme (RPS-S) is retrieved from memory for which its $n$–th unfolding (T-S) results in a "maximal similarity" to the current (target) problem. The source program scheme can either be a concrete program with primitive operations or an already abstracted scheme. The source scheme is modified with respect to the mapping obtained between its unfolding and the target. Modification can involve a simple re-instantiation of primitive symbols or more complex adaptations. Finally, the source scheme and the new program are generalized to a more abstract scheme. The new RPS and the abstracted RPS are stored in memory.

After introducing our approach to anti-unification, all components of programming by analogy are discussed. An overview of our work on constructing programming by analogy algorithms is given in appendix A13.

## 2 RESTRICTED 2ND–ORDER ANTI–UNIFICATION

### 2.1 RECURSIVE PROGRAM SCHEMES REVISITED

A program is represented as a recursive program scheme (RPS) $\mathcal{S} = (\mathcal{G}, t_0)$, as introduced in definition 7.17. The main program $t_0$ is defined over a term algebra $\mathcal{T}_\Sigma(X)$ where signature $\Sigma$ is a set of function symbols and $X$ is a set of variables (def. 7.1). In the following, we split $\Sigma$ in a set of primitive function symbols $F$ and a set of user-defined function names $\Phi$ with $F \cap \Phi = \emptyset$ and $\Sigma = F \cup \Phi$. A recursive equation in $\mathcal{G}$ consists of a function head $G(x_1, \ldots, x_m)$ and a function body $t$. The function head gives the name of the function $G \in \Phi$ and the parameters of the function with $X = \{x_1, \ldots, x_m\}$. The function body is defined over the term algebra, $t \in \mathcal{T}_\Sigma(X)$. The body $t$ contains the symbol $G$ at least once. Currently, our generalization approach is restricted to RPSs where $\mathcal{G}$ contains only a single equation and where $t_0$ consists only of the call of this equation.

Since program terms are defined over function *symbols*, in general, an RPS represents a class of programs. For program evaluation, all variables in $X$ must be instantiated by concrete values, all function symbols must be interpreted by executable functions, and all names for user-defined functions must be defined in $F$.

In the following, we introduce special sets of variables and function symbols to discriminate between concrete programs and program schemes which were generated by anti-unification. The set of variables $X$ is divided into variables $X_p$ representing input variables and variables $X_{au}$ which represent first-order generalizations, that is, generalizations over first-order constants. The set of function symbols $F$ is divided into symbols for primitive functions $F_p$ with a fixed interpretation (such as *+(x, y)* representing addition of two numbers) and function symbols with arbitrary interpretation $\Phi_{au}$. Symbols in $\Phi_{au}$ represent function variables which were generated by second-order generalizations, that is generalizations over primitive functions. Please note, that these function variables do *not* correspond to names of user-defined functions $\Phi$!

**Definition 12.1 (Concrete Program)** *A concrete program is an RPS over $\mathcal{T}_\Sigma(X_p)$ with $\Sigma = F_p \cup \Phi$. That is, it contains only input variables in $X_p$, symbols for primitive functions in $F_p$ and function calls $G_i \in \Phi$ which are defined in $\mathcal{G}$.*

**Definition 12.2 (Program Scheme)** *A program scheme is an RPS defined over $\mathcal{T}_\Sigma(X_p \cup X_{au})$ with $\Sigma = F_p \cup \Phi_{au} \cup \Phi$. That is, it can contain input variables in $X_p$, symbols for primitive functions in $F_p$, function calls $G_i \in \Phi$ which are defined in $\mathcal{G}$, as well as object variables $y \in X_{au}$, and function variables $\chi \in \Phi_{au}$.*

An RPS can be unfolded by replacing the name of a user-defined function by its body where variables are substituted in accordance with the function call as introduced in definition 7.29. In principle, a recursive equation can be unfolded infinitely often. Unfolding terminates, if the recursive program call is replaced by $\Omega$ (the undefined term) and the result of unfolding is a finite program term (see def. 7.19) $T \in \mathcal{T}_\Sigma(X_p \cup X_{au})$ with $\Sigma = F_p \cup \Phi_{au} \cup \{\Omega\}$. That is, $T$ does not contain names of user-defined functions $G_i \in \Phi$.

## 2.2 ANTI-UNIFICATION OF PROGRAM TERMS

First order anti-unification was introduced in section 1.2 in chapter 7. In the following, we present an extension of Huet's declarative, first-order algorithm for anti-unification of pairs of program terms (Huet, 1976; Lassez et al., 1988) by introducing three additional rules. The term that results from the anti-unification of two terms is their most specific anti-instance.

**Definition 12.3 (Instance/Anti-Instance)** *If $t_1 \ldots t_n$ and $u$ are terms and for each $t_i$ there exists a substitution $\sigma_i$ $(1 \leq i \leq n)$ such that $t_i = u\sigma_i$, then the terms $t_i$ are* instances *of $u$ and $u$ is an* anti-instance *of the set $\{t_1, \ldots, t_n\}$.*

*$u$ is the* most specific anti-instance *of the set $\{t_1, \ldots, t_n\}$ if, for each $u'$ which is also an anti-instance of $\{t_1, \ldots, t_n\}$, there exists a substitution $\theta$ such that $u' = u\theta$.*

Simply spoken, an anti-instance reflects *some* commonalities shared between two or more terms, whereas the most specific anti-instance reflects *all* commonalities.

Huet's algorithm constructs the most specific anti-instance of two terms in the following way: If both terms start with the same function symbol, this function symbol is kept for the anti-instance, and anti-unification is performed recursively on the function arguments. Otherwise, the anti-instance of the two terms is a variable determined by an injective mapping $\varphi$. $\varphi$ ensures that each occurrence of a certain pair of sub-terms within the given terms is represented by the same variable in the anti-instance.

According to Idestam-Almquist (1993), $\varphi$ can be considered a *term substitution*:

**Definition 12.4 (Term Substitution)** *A finite set of the form*

$$[\![(t_1, u_1)/x_1, \ldots, (t_n, u_n)/x_n]\!]$$

*is a* term substitution, *iff $[\![x_1/t_1, \ldots, x_n/t_n]\!]$ and $[\![x_1/u_1, \ldots, x_n/u_n]\!]$ are substitutions, and the pairs $(t_1, u_1), \ldots, (t_n, u_n)$ are distinct.*

Huet's algorithm only computes the most specific *first-order* anti-instance. But in order to capture as much of the common structure of two programs

*Table 12.1.* A Simple Anti-Unification Algorithm

**Function Call**: $\mathbf{au}(t_1, t_2)$ with terms $t_1, t_2 \in M(V, F \cup \Phi)$
**Initialization**: term substitutions $\varphi = \emptyset$
**Rules:**

- **Same-Term**: $\mathbf{au}(t, t) = t$
- **Var-Term**: $\mathbf{au}(t_1, t_2) = y$ with $\varphi = \varphi \circ [\![t_1, t_2/y]\!]$ (where either $t_1 \in X$ and $t_2 \in \mathcal{T}_\Sigma(\mathcal{X})$ or $t_1 \in \mathcal{T}_\Sigma(X)$ and $t_2 \in X$
- **Same-Function**: $\mathbf{au}(f(x_1, \ldots, x_n), f(x'_1, \ldots, x'_n)) = f(\mathbf{au}(x_1, x'_1), \ldots, \mathbf{au}(x_n, x'_n))$
- **Same-Arity**: $\mathbf{au}(f(x_1, \ldots, x_n), g(x'_1, \ldots, x'_n)) = \chi(\mathbf{au}(x_1, x'_1), \ldots, \mathbf{au}(x_n, x'_n))$ with $\varphi = \varphi \circ [\![f, g/\chi]\!]$
- **Diff-Arity**: $\mathbf{au}(f(x_1, \ldots, x_n), g(x'_1, \ldots, x'_m)) = y$ with $\varphi = \varphi \circ [\![f(x_1, \ldots, x_n), g(x'_1, \ldots, x'_m)/y]\!]$ (where $n \neq m$)

as possible in an abstract scheme we need at least a second order (function) mapping. In contrast to first order approaches, higher order anti-unification (as well as unification) in general has no unique solution – that is, a notion of *the* most specific anti-instance does not exist – and cannot be calculated efficiently (Siekmann, 1989; Hasker, 1995).

Therefore, we developed a very restricted second-order algorithm. Its main extension compared with Huet's is the introduction of function variables for functions of the same arity.

Based on the results obtained with this algorithm, we plan careful extensions, maintaining uniqueness of the anti-instances and efficiency of calculation. A more powerful approach to second order anti-unification was proposed for example by Hasker (1995), as mentioned in chapter 10.

Our algorithm is presented in table 12.1. The **Var-Term**, **Diff-Arity**, and **Same-Function** rules correspond to the two cases of Huet's algorithm. **Same-Term** is a trivial rule which makes the implementation more efficient. Our main extension to Huet's algorithm is the **Same-Arity** rule, by which function variables $\chi \in \Phi_{au}$ are introduced for functions of the same arity. Huet's termination case has been split up into two rules (**Var-Term** and **Diff-Arity**) because the **Diff-Arity** rule can be refined in future versions of this algorithm to allow generalizing over functions with different arities as well.

Proofs of uniqueness, correctness, and termination, are given in (Sinha, 2000).

To illustrate the algorithm, we present a simple example: Let the two terms which are input to $\mathbf{au}(t_1, t_2)$ be

$$t_1 = if(eq0(x), 1, *(x, fac(p(x))))$$
$$t_2 = if(eq0(z), 0, +(z, sum(p(z))))$$

Then,

$$\mathbf{au}(t_1, t_2) = if(eq0(y_1), y_2, \chi_2(y_1, \chi_1(p(y_1))))$$

with

$$\varphi = [\![y_1/(x, z),\ y_2/(1, 0),\ \chi_1/(fac, sum),\ \chi_2/(*, +)]\!]$$

as their most specific anti–instance. The rules used are: **Same-Function** (twice), **Var-Term**, **Same-Arity** (twice), **Var-Term** (making use of $\varphi$), **Same-Arity**, **Same-Function**, and finally **Var-Term** (again, with $\varphi$). In table 12.4, additional examples for anti-unification are given.

# 3 RETRIEVAL USING TERM SUBSUMPTION

## 3.1 TERM SUBSUMPTION

By using subsumption of anti-instances, retrieval can be based on a qualitative criterium of structural similarity (Plaza, 1995) instead of on a quantitative similarity measure. The advantage of a qualitative approach is that it is parameter-free. That is, there is no need to invest in the somewhat tedious process of determining "suitable" settings for parameters and no threshold for similarity must be given.

The subsumption relation is defined usually only for *clauses* (see sect. 3.3 in chap. 6). This definition is based either on the subset/superset relationship – which exists between clauses since they can be viewed as sets (the conjunction of literals is commutative) – or on the existence of a substitution which transforms the more general (subsuming) clause into the more special (subsumed) one. Since terms cannot be viewed as sets, and therefore the subset/superset relationship does not exist, our definition of term subsumption has to be based on the existence of substitutions only.

**Definition 12.5 (Term Subsumption)** *A term $t_1$ subsumes a term $t_2$ (written $t_2 \leq t_1$) iff there exists a substitution $\theta$ such that $t_1\theta = t_2$.*

Note, that $\theta$ must be a "proper" substitution, not a term substitution (def. 12.4).

Substitution $\theta$ is obtained by unifying $t_1$ and $t_2$. If $\theta$ is empty and $t_1 \neq t_2$, unification of $t_1$ and $t_2$ has failed. In that case, the subsumption relation between $t_1$ and $t_2$ is undecidable. That is, the subsumption relation does not

*Table 12.2.* An Algorithm for Retrieval of RPSs

Given

$T$ – a finite program.

$\mathcal{CB} = \{R_1, \ldots, R_n\}$ – a case base which is a set (represented as a list) of RPSs $\mathcal{S}_i$.

$\mathcal{U}$ – a set (initially empty) of most specific anti-instances.

$\forall R_i \in \mathcal{CB}$:

- Let $T_i = \mathbf{unfold}(\mathcal{S}_i)$. (unfold $\mathcal{S}_i$ according to def. 7.29 until the resulting term $T_i$ has at least the length of $T$)
- Let $u_i = \mathbf{au}(T, T_i)$. (anti-unify $T$ and $T_i$)
- If $\exists u \in \mathcal{U}$ with $u_i \approx u$ or $u \leq u_i$ then $\mathcal{U}$.
- $\forall u \in \mathcal{U}$ with $u_i \leq u$: $\mathcal{U} := (\mathcal{U} \setminus u) \cup u_i$
- Return the RPSs $\mathcal{S}_i$ associated with all $u_i \in \mathcal{U}$.

define a total but a partial order over terms. For two isomorphic terms, that is, terms which can by unified by variable renaming only, we write $t_1 \approx t_2$.[2]

Our algorithm for retrieving the "maximally similar" RPSs from a linear case base is given in table 12.2. In general, the algorithm returns a *set* of possible source programs rather than a unique source due to the undecidability of subsumption for non-unifiable terms. Input to the algorithm is a finite program $T$. For each RPS in memory, this RPS $\mathcal{S}_i$ is unfolded to a term $T_i$ such that it has at least the length of $T$ and the most specific anti-instance $u_i$ of $T$ and $T_i$ is calculated. This anti-instance is only inserted in the set of anti-instances $\mathcal{U}$ if it is either subsumed (i. e., is more special) by an anti-instance $u \in \mathcal{U}$ or if it is not unifiable with any $u \in \mathcal{U}$. Furthermore, all anti-instances in $\mathcal{U}$ which subsume (i. e., are more general) than $u_i$ are removed from $\mathcal{U}$. The algorithm returns all RPSs $\mathcal{S}_i$ from the case-base which are associated with an $u_i$ in the final set of anti–instances $\mathcal{U}$.

## 3.2 EMPIRICAL EVALUATION

We presented terms to six subjects (all of whom were familiar with functional programming) asking them to identify similarities among the terms. All these terms were RPSs unfolded twice. We chose ten RPSs which we considered "relevant". The RPSs represented various recursion types, such as linear recursion (`SUM, FAC, APPEND, CLEARBLOCK`), tail recursion (`MEMBER,`

[2]Note, that here we speak of constructing an order over terms and isomorphism is defined with respect to the *unification* of terms. Later in this chapter, we will discuss isomorphism in the context of anti-unification, that is generalization, of terms.

*Table 12.3.* Results of the similarity rating study

| Goal RPS | Subjects' Choice | Majority | Program Returns |
|---|---|---|---|
| REVERT | 4: 33%, 2, 3, 5, 9: 17% each | – | {7, 8, 10} |
| SUM | 5: 83%, 10: 17% | 5 | {**5**, 10} |
| CLEARBLOCK | 6: 100% | 6 | {**6**} |
| GGT | 1: 67%, 6: 33% | 1 | {8, 10} |
| FAC | 2: 100% | 2 | {**2** ,8 ,9} |
| APPEND | 3: 83%, 2: 17% | 3 | {1, **3**, 7} |
| MEMBER | 10: 33%, 1, 2, 4, 6: 17% each | – | {1, 9} |
| LAH | 10: 50%, 9: 33%, 6: 17% | – | {9, 10} |
| BINOM | 10: 100% | 10 | {7, 8, **10**} |
| FIBO | 9: 50%, 8: 33%, 3: 17% | – | {2, 8, 9} |

REVERT), tree recursion (BINOM, FIBO, LAH), and a function calling a further function (GGT calling MOD). All functions are given in appendix C8.

In each rating task the subjects were presented one term and a "case base" consisting of the other nine terms. The order, in which the case base was presented, varied among subjects. They were requested to choose the term from the case base which was most similar to the given term.

In table 12.3, the results of this rating study are shown.

The rightmost column shows the results of our algorithm. Only for the CLEARBLOCK problem (3) the algorithm yields a unique result. In all other cases, a set of "most similar" terms is returned. This is due to both subsumption equivalences and the fact that subsumption is not always decidable (see section 3.1).

The RPS preferred by a majority (two-thirds or more) of the subjects was always among the RPSs returned by the algorithm (except for GGT). Moreover, there are symmetries in retrieval (e. g., FAC and SUM) that can be explained by a common recursion scheme of these RPSs. Generally, at least one RPS of the same recursion type as the goal RPS is retrieved, provided that there is such an RPS in the case base.

## 3.3 RETRIEVAL FROM HIERARCHICAL MEMORY

We demonstrated retrieval of concrete programs or abstract schemes for the case of a linear memory. Memory can be organized more efficiently by introducing a hierarchy of programs. That is, for each pair of programs which were used as source and target in programming by analogy, their anti-instance is introduced as parent.

For hierarchical memory organization, retrieval can be restricted to a subset of the memory in the following way: Given a new finite program term $t$,

- identify the most specific anti-instance $\mathbf{au}(T, T_i)$ for all unfolded RPSs which are root-nodes,
- in the following only investigate the children of this node.

As in the linear case, in general, there might not exist a unique minimal anti-instance, that is, more than one tree in the memory must be explored. Furthermore, retrieval from hierarchical memory is based on a monotonicity assumption: If a program term $T_i$ is less similar to the current term $T$ than a program term $T_j$ then the children of $T_j$ cannot be more similar to $T$ than $T_i$ and the children of $T_i$. While this assumption is plausible, it is worth further investigation. Our work on memory hierachization is work in progress and we plan to provide a proof for monotonicity together with an empirical comparison of the efficiency of retrieval from hierarchical versus linear memory.

# 4 GENERALIZING PROGRAM SCHEMES

There are different possibilities, to obtain an RPS $\mathcal{S}_{1,2}$ which generalizes over a pair of RPSs $\mathcal{S}_1$ and $\mathcal{S}_2$:

- The generalized program term $T_{1,2}$ which was constructed as $\mathbf{au}(T_1, T_2)$ with $T_1 = \mathbf{unfold}(\mathcal{S}_1)$ and $T_2 = \mathbf{unfold}(\mathcal{S}_2)$ can be folded, using our standard approach to program synthesis described in chapter 7.
- The term-substitution $\varphi$, obtained by constructing $\mathbf{au}(T_1, T_2)$, can be extended to $\varphi' = \varphi \circ [G_1, G_2/\chi_{1,2}]$, that is, by introducing a variable for the names of the recursive functions. Then, for our restricted approach dealing with functions of different arity in a first-order way, it is enough to apply $\varphi'$ to either $\mathcal{S}_1$ or $\mathcal{S}_2$. That is, for term-substitutions $\varphi' = \{(T_{1,i}, T_{2,i}/y_i) \mid i = 1..n\}$ where $y_i \in X_{au} \cup \Phi_{au}$, with projections $\sigma_1 = \{(T_{1,i}/y_i)\}$, $\sigma_2 = \{(T_{2,i}/y_i)\}$ it holds, that $\sigma_1(\mathcal{S}_1) = \sigma_2(\mathcal{S}_2)$.
- The anti-unification algorithm defined above can be applied to recursive equations $G_i(x_1, \ldots, x_n) = t_i$, if the equations are reformulated as equality terms: `(= (Gi x1 ... xn) ti)`.

All three possibilities have their advantages: Folding of abstracted terms can be used to check whether the obtained generalization really represents