Université Catholique de Louvain
Faculté des Sciences Appliquées

Unité d'Informatique

Logic Algorithm Synthesis from Examples and Properties

Pierre Flener

June 1993

Thèse présentée en vue
de l'obtention du grade de
Docteur en Sciences Appliquées

Logic Algorithm Synthesis from Examples and Properties

Pierre Flener

June 1993
Thèse présentée en vue
de l'obtention du grade de
Docteur en Sciences Appliquées

Université Catholique de Louvain
UNITE D'INFORMATIQUE
Place Sainte-Barbe, 2
B-1348 Louvain-la-Neuve
Belgium

Abstract

In the context of logic programming-in-the-small, we propose logic specifications by examples and properties, the latter being disambiguating generalizations of the examples. Such specifications are easy to elaborate, but are also usually incomplete, in the sense that the intentions need not (or cannot) be fully described. Algorithm synthesis then consists of semi-automatically extrapolating these intentions, and of designing a (recursive) algorithm implementing them. We develop a very disciplined approach to algorithm synthesis, namely stepwise instantiation of the place-holders of a divide-and-conquer algorithm schema. Moreover, rather than using a uniform method for these instantiations, we deploy for each place-holder the best-suited method from a generic tool-box of (deductive, inductive, …) methods. Special care is taken to handle the correctness and progression aspects of the synthesis.

Keywords

Automatic programming, program synthesis, logic programming, intentions, specification, algorithm, algorithm design, algorithm schema, algorithm correctness, algorithm comparison, stepwise design, inductive logic programming, machine learning, inductive inference, generalization, deductive inference, automated theorem proving.

Acknowledgments

I hereby express my deepest gratitude to my advisor, Prof. Yves Deville. His patience and encouraging comments were music in my ears whenever I thought I had painted myself into a corner, and his expertise and assistance were invaluable during the development of some theoretical aspects. When deadlines were close, his availability beyond the call of duty was priceless. Moreover, his relentless pursuit of excellence drove me to improvements and generalizations I would never have dreamt of, and his insightful overall perception of my research area guided me to a better understanding of my results and contributions. Finally, he tempered my exuberant and provocative writing style into a cautious, diplomatic language. Thank you, Yves!

My warmest thanks to Prof. Alan W. Biermann (Duke University, Durham, NC, USA), who accepted my application for a leave-of-absence at his department, and who later agreed on extending this research stay and on adding other, shorter stays. His faith in my approach kept my spirit alive during many months of darkness. As my mentor on program synthesis research, he distilled a lot of wisdom during our numerous discussions.

I acknowledge extraordinary assistance by Prof. Baudouin Le Charlier (FUNDP, Namur, Belgium). His spiritual patronage on this thesis is unmistakable, and the fruit of many interesting discussions. My gratefulness further goes to Prof. Axel van Lamsweerde (UCL), who awakened my passion for research as my MS advisor, and who provided many constructive comments on earlier versions of this dissertation. Prof. Michel Sintzoff and Prof. Elie Milgrom (both at UCL) also offered useful feedback during the writing process.

Many thanks to Prof. Norbert Fuchs (Universität Zürich, Switzerland) and Prof. Klaus-Peter Jantke (TH Leipzig, Germany) for their interest in my research, and for inviting me abroad to give seminars on it. Prof. Laurent Fribourg (ENS, Paris, France) kindly assisted me during the development of some aspects that are based on his own research. Moreover, I'm indebted to my fellow PhD-student veterans Luboš Popelinský (Masaryk University of Brno, Czech Republic) and Tim Gegg-Harrison (Duke University, NC, USA), for many productive life and e-mail discussions.

The research for this thesis was done at three universities, and hence a possibly unusually long list of miscellaneous acknowledgments:

From October 1988 to December 1991, I was a member of the *Folon Research Project* at the *Institut d'Informatique* of the *Facultés Universitaires Notre-Dame de la Paix* (FUNDP), in Namur (Belgium). Many thanks to my Folon colleagues (by chronological order of arrival) Pierre Deboeck, Jean Burnay, Jean-Pierre Hogne, Pascal van Hentenryck, Bruno Desart, and Jean Henrard for inspiring discussions, and to our office-mates Kaninda Musumbu and Aziz Mounji for contributing to the good spirit in our office. Thanks also to the systems guys Bruno Delcourt, Philippe Du Bois, and Naji Habra, for keeping backups handy when I goofed. The Folon project was supported by the *Services de Programmation de la Politique Scientifique* (SPPS) of the *Belgian Government*, under grant RFO/AI/12.

From January 1990 to March 1991, I was a visiting research scientist at the *Department of Computer Science* at *Duke University*, in Durham (North Carolina, USA). I very much appreciated sharing an office with Richard Hipp, Margie Dietz, and Steve Owen, and I actually feel indebted to the entire *Butterfly Wing* for the excellent atmosphere. Thanks also to the systems people Karen Feldman and Jeff Tannehill, for their kind assistance. This stay was also supported by the SPPS, namely under grant RFO/AI/12/2.

From January 1992 to now, I was a PhD candidate at the *Unité d'Informatique* of the *Université Catholique de Louvain* (UCL), in Louvain-la-Neuve (Belgium). Many thanks to my office-mates (by chronological order of arrival) Simone Amerijckx, Marie-Hélène Liégeois, Pierre-Yves Schobbens, Philippe de Groote, Bruno Charlier, Frédéric Geurts, and Vincent Lombart, for delightful days of research. Thanks also to the systems people Hubert Broze, Marie-France Declerfayt, and Francis Degey, for their efficient trouble-shooting. My gratefulness further goes to Prof. Elie Milgrom for his confidence in my research. Since May 1992, I am supported by the *Ministry of Scientific Research and Cultural Affairs* of the *Government of Luxembourg*, under grant BFR 92/017.

Many people at these three institutions, and elsewhere, got more or less close looks at my research. During my various local seminars, I very much appreciated the perceptive comments made by Prof. Don W. Loveland, Prof. Gopalan Nadathur, Tim Gegg-Harrison, and Richard Hipp (all at Duke University), and by Dr. Naji Habra, Dr. Jean-Marie Jacquet, and Prof. Pierre-Yves Schobbens (all at FUNDP). My gratitude also to those who—more or less—volunteered to review preliminary versions of some chapters of this thesis, namely Bruno Charlier (UCL) and Luboš Popelinský (Masaryk University of Brno). Many thanks to the anonymous referees of my papers, for showing me what I should explain with more care. Finally, I thank all those whom I might have forgotten to mention, but who feel that they have had a direct or indirect impact on this thesis.

I would also like to commend *Frame Technology Corporation* (San Jose, California, USA) for developing the *FrameMaker* desktop publishing system (which suits my needs so much), for their friendly customer assistance, and especially for giving me a free license during the six months where I had access to none.

Finally, extra credit goes to my parents, siblings, relatives, colleagues, and friends, for encouraging me throughout these years. Without you, I wouldn't have had the stamina to go through all this. A special tribute to my friends Marc Blancart and Guy Huys, for helping me to think of other things, for instance while accompanying me on backpacking journeys, and to Jonathan Polito, for many "Zen-ish" bike-rides. A moral assist goes to Duke's Blue Devils basketball teams, to the numerous volley-ball teams I was part of, to the Alternative Happy Hour crew, and to the members of the International Association at Duke University.

Background, Terminology, and Conventions

This research is set within the framework of logic programming. We assume the reader is familiar with this background (see, for instance, [Lloyd 87], of which we adopt the terminology, unless otherwise noted). Here follows a list of the most used typographic conventions, abbreviations, and predefined symbols. Others will be introduced as necessary, and the entire list can be found in the Appendix.

Typographic conventions

Function symbols (functors) and predicate symbols start with a lower-case letter. We often write them out followed by a slash "/" and their arity. Examples are $a/0$, $f/2$, and $p/3$. Exceptions are some primitive symbols, such as $=/2$, $</2$, and $\bullet/2$. Functors of arity 0 are called *constants*. By abuse of language, we often say "predicate" instead of "predicate symbol". Variable symbols, function variable symbols, and predicate variable symbols start with an upper case letter. Examples are *X*, *F*, and *P*. An anonymous variable is denoted by an underscore "_". Schema variable symbols and notation variable symbols consist of a lower-case letter. Examples are *i, j, k, m*, and *n*. The distinction with one-letter constants should always be obvious from context. By abuse of language, we often say "variable" instead of "variable symbol".

Terms and atoms are usually written in prefix notation. If no ambiguity arises, unary terms and atoms are sometimes written without parentheses, while binary terms and atoms are sometimes written in infix notation.

A constructed list with head H and tail T is denoted $H\bullet T$, whereas *nil* denotes the empty list. Another notation for a constructed list with head H and tail T is $[H|T]$, whereas [] is another notation for the empty list. These alternative notations allow shorthands such as $[H_1,H_2,\ldots,H_n]$ for $\bullet(H_1,\bullet(H_2,\bullet(\ldots,\bullet(H_n,nil)\ldots)))$, and $[H_1,H_2,\ldots,H_n|T]$ for $\bullet(H_1,\bullet(H_2,\bullet(\ldots,\bullet(H_n,T)\ldots)))$, where $n>0$.

Non-negative integers are successors of the constant zero, which is denoted 0. The sequence of integers is $0, s(0), s(s(0)), \ldots, s^n(0), \ldots$, where $s/1$ is called the successor functor. Shorthands are $0, 1, 2, \ldots, n, \ldots$, respectively.

Variable symbols, predicate variable symbols, functors, and predicate symbols within text paragraphs are written in *Times-italics*. However, entirely formalized paragraphs, such as specifications, (logic) algorithms, and (logic) programs, are written in `Courier`.

Term vectors and variable vectors of indeterminate, but finite, length are denoted by **boldface** symbols. Examples are $\boldsymbol{t}$ and $\boldsymbol{X}$. Vectors of vectors of indeterminate, but finite, length are denoted by **underlined boldface** symbols. Examples are $\underline{\boldsymbol{t}}$ and $\underline{\boldsymbol{X}}$. Given an integer n, an n-tuple of length n is written using angled brackets. An example is $\langle t_1,t_2,\ldots,t_n\rangle$. Note that an n-tuple is a term, whereas a vector is not a term.

Names of sets or relations start with an upper-case letter, and are written in *Zapf*. An example is $\mathcal{R}$. Exceptions are some primitive symbols, such as $=/2$, $</2$, and $\in/2$. The predicate symbol corresponding to a relation is the name of the relation, but it then starts with a lower-case letter, and is written in *Times-italics*. For instance, *r* is the predicate symbol for relation $\mathcal{R}$. The complement of a relation has the same name as the relation itself, but crossed out by a slash "/", if the name is a primitive symbol, and $\overline{\text{overlined}}$, otherwise.

The binding of a term t to a variable X is denoted X/t. Substitutions are denoted by Greek lower-case letters. Examples are σ, ρ, and θ.

The construct $F[\boldsymbol{X}]$ denotes a well-formed formula F whose free variables are $\boldsymbol{X}$. The construct $F[\boldsymbol{t}]$ then denotes $F[\boldsymbol{X}]$ where the free occurrences of $\boldsymbol{X}$ are replaced by the terms $\boldsymbol{t}$. The **boldface** construct ***r(s,t)*** denotes a finite conjunction $r(s_1,t_1) \wedge r(s_2,t_2) \wedge \ldots \wedge r(s_n,t_n)$.

The end of a multi-paragraph example is indicated by a black diamond: ♦.
The end of a proof is indicated by a quad: ❑.
Emphasized words are underlined. Defined words are in *italics*.

Abbreviations

General abbreviations

iff	if and only if
wrt	with respect to

Scientific abbreviations

BNF	Backus-Naur Form
gci	greatest common instance
glb	greatest lower bound
lub	least upper bound
mgu	most general unifier
msg	most specific generalization
NF	Negation-as-Failure
SL resolution	Linear resolution with Selection Function
SLD resolution	SL resolution for Definite clauses
SLDNF resolution	SLD resolution with the NF rule
wff	well-formed formula
wfr	well-founded relation

Glossary of Symbols

Sets

$\mathcal{A}$	the set of atoms constructed from $\mathcal{Q}$ and $\mathcal{T}$
$\mathcal{B}$	the Herbrand base (that is the set of ground atoms constructed from $\mathcal{Q}$ and $\mathcal{U}$); $\mathcal{B} \subseteq \mathcal{A}$
$\mathcal{C}$	the set of predefined base case constants of inductively defined data types; $\mathcal{C} \subseteq \mathcal{F}$; $\mathcal{C}$ is here assumed to be $\{0, nil\}$
$\mathcal{F}$	the set of functors
$\mathfrak{I}$	the intended interpretation
$\mathcal{Q}$	the set of predicate symbols
$\mathcal{R}$	the intended relation
$\mathcal{T}$	the set of terms constructed from $\mathcal{F}$ and $\mathcal{V}$
$\mathcal{U}$	the Herbrand universe (that is the set of ground terms constructed from $\mathcal{F}$); $\mathcal{U} \subseteq \mathcal{T}$
$\mathcal{V}$	the set of variable symbols
$\mathcal{W}$	the set of wff constructed from $\mathcal{A}$ and $\mathcal{V}$

Constants

nil or []	the empty list
ω	infinity
0	the integer zero
∅ or {}	the empty set

Functors of arity n, where $n>0$

cons(E)	the set of constants occurring in expression E
dom(σ)	the domain of substitution σ: *dom*(σ) ⊆ $\mathcal{V}$
funct(*E*)	the set of functors occurring in expression E
msg(s,t)	the most specific generalization of terms s and t
range(σ)	the range of a substitution σ: *range*(σ) ⊆ $\mathcal{T}$

s(i)	the successor of integer *i*
vars(E)	the set of variables occurring in expression *E*
$\#\mathcal{S}$	the number of elements of set $\mathcal{S}$, or of vector $\mathcal{S}$
H•T	the list constructed of head *H* and tail *T*
$\mathcal{S}_1 \cup \mathcal{S}_2$	the union of the sets $\mathcal{S}_1$ and $\mathcal{S}_2$
$\mathcal{S}_1 \cap \mathcal{S}_2$	the intersection of the sets $\mathcal{S}_1$ and $\mathcal{S}_2$
$\mathcal{S}_1 \setminus \mathcal{S}_2$	the difference of the sets $\mathcal{S}_1$ and $\mathcal{S}_2$

Primitive predicates

false/0	never holds
true/0	always holds
$s = t$	term *s* is unifiable with term *t*
$s \leq t$	term *s* is less general than term *t*, or integer *s* is less than or equal to integer *t* (according to context)
$e \in \mathcal{S}$	term *e* is an element of set $\mathcal{S}$
$\mathcal{S}_1 \subseteq \mathcal{S}_2$	set $\mathcal{S}_1$ is a subset of set $\mathcal{S}_2$

Connectives for well-formed formulas

$\forall$	for all (universal quantification)
$\exists$	there is (existential quantification)
$\neg$	not (negation)
$\vee$	inclusive or
$\veebar$	exclusive or
$\wedge$	and
$\Rightarrow$	implies
$\Leftarrow$	if
$\Leftrightarrow$	if-and-only-if
$\vee_{a \leq i \leq b} F_i$	$F_a \vee F_{a+1} \vee \ldots \vee F_b$, if $b \geq a$, and *false* otherwise
$\wedge_{a \leq i \leq b} F_i$	$F_a \wedge F_{a+1} \wedge \ldots \wedge F_b$, if $b \geq a$, and *true* otherwise

Connectives for logic programs

,	and
$\leftarrow$	if

Meta-logical connectives

$\models$	Herbrand-logical consequence
$\vdash$	derivability

Precedence hierarchy (highest-to-lowest) of the wff connectives

$\neg, \forall, \exists$

$\vee, \veebar$

$\wedge$

$\Leftarrow, \Rightarrow, \Leftrightarrow$

TABLE OF CONTENTS

II BUILDING BLOCKS

Introduction

Automatic programming (also called program synthesis) is an often suggested solution to the software crisis, which has been haunting commercial software production for over two decades now. Indeed, if we could write a program that develops correct programs from specifications, with as much (or as little) interaction as the specifier wants, then the dreaded program testing and program maintenance stages would disappear from the software life-cycle, and one could instead focus on the more creative tasks of specification elaboration, testing, and maintenance, because replay of program development would become less costly.

Understanding the Title of this Thesis

The framework of this thesis is algorithm synthesis from "incomplete" specifications. More precisely, we tackle the problem of "*logic algorithm synthesis from specifications by examples and properties*". This formulation implies several things. First, the chosen programming paradigm is *logic* programming. Second, we are here only interested in synthesizing recursive logic programs, which we call logic *algorithms*. Moreover, the focus is on the actual algorithm *synthesis*, and on the declarative semantics of such logic algorithms, but not on their transformation, optimization, or implementation. Third, synthesis starts from incomplete *specifications*, in the sense that one deliberately admits a lack of information in the specification with respect to the intentions: synthesis is meant to extrapolate these intentions. The chosen specification formalism is a compromise between specifications by examples and axiomatic specifications: *examples* of the intended relation are given, as well as *properties*, which are a specialized form of axioms, but also a generalized form of examples. Properties are meant to disambiguate examples. This approach should allow faster and more reliable synthesis than from examples alone, but also more natural and understandable specifications than first-order logic axiomatizations.

The Claim of this Thesis

These specification and programming paradigms have been chosen to illustrate our claim that "*program synthesis can be effectively performed by successively filling in the place-holders of some algorithm schema, each such instantiation being done by deploying the best-suited method of a generic tool-box of (inductive, deductive, …) methods*". Again, this formulation implies several things. First, we require a firm theoretical grip on the *effectiveness* of the synthesis: the development and application of suitable correctness criteria is a central theme in this thesis. Second, we want to decompose synthesis into a *succession* of well-defined software engineering tasks: this is where the notion of *algorithm schema* comes in. We here exclusively focus on the divide-and-conquer schema and define its *place-holders*. Schema-guidance is a very powerful way to inject algorithm knowledge into synthesis, and thus reduce search spaces. Third, we suggest that synthesis need not be done by a unique mechanism in one fell swoop, nor by a decomposition into sub-tasks that are performed using a single kind of reasoning: indeed, a *generic tool-box* of methods can be developed, each *method* being able to *instantiate* place-holders of schemas. We develop such a tool-box: some methods perform *inductive* reasoning, other methods perform *deductive* reasoning, yet other methods simply retrieve instances from databases.

Organization of this Thesis

This thesis is divided into three parts: Part I is about the state-of-the-art in automatic programming, Part II provides some building blocks for a synthesis mechanism, and Part III describes our synthesis mechanism in terms of these building blocks.

Part I: *State of the Art*

In Part I, we present the *state-of-the-art* of the topics underlying this thesis, and introduce the relevant terminology along the way. The idea is to gradually introduce the objectives of this research by narrowing in from more general aims. Note, however, that Chapters 2 to 4 may be read in any order. This state-of-the-art is unusually long, because this thesis is at the intersection of numerous domains, such as software engineering, deductive inference and automated theorem proving, inductive inference and machine learning, and algorithm design methodologies.

In Chapter 1, we give a general introduction to the field of *automatic programming*. After a presentation of the grand aim of program synthesis research, we discuss the languages used for expressing specifications, algorithms, and programs. We then propose a classification scheme for synthesis mechanisms, and conclude with our personal viewpoint on the requirements and promises of automatic programming research.

In Chapter 2, we survey the use of *deductive inference in automatic programming*. Axiomatic specifications are expected to be complete and non-ambiguous, but are usually also quite lengthy and artificial. Deductive synthesis from axiomatic specifications can be classified into transformational synthesis, proofs-as-programs synthesis (or constructive synthesis), and schema-guided synthesis. We survey the achievements of deductive synthesis of LISP functions and of Prolog predicates.

In Chapter 3, we survey the use of *inductive inference in automatic programming*. Specifications by examples are concise and natural, but are usually also incomplete and ambiguous, due to the insufficient expressive power of examples. As inductive inference is much less known than deductive inference, we first survey this field. Inductive synthesis from specifications by examples can be classified into trace-based synthesis and model-based synthesis. We survey the achievements of inductive synthesis of LISP functions and of Prolog predicates.

In Chapter 4, we summarize a *methodology of logic program development*. The original promise of programming in first-order logic when using Prolog is indeed impaired by Prolog's incomplete and unsound inference engine, and by the availability of non-logical predicates. But these features were deliberately chosen so as to make Prolog a practical programming language. Hence there is a need for a language-independent logic programming methodology that assists in reconciling the gap between the declarative and the procedural semantics. Such a methodology has been formulated by [Deville 87, 90]. It aims at programming-in-the-small, and is meant for algorithmic problems. It is divided into three stages: (1) elaboration of an informal specification; (2) design of a "logic algorithm" (an algorithm expressed in logic), and possibly its transformation; and (3) derivation of a logic program, and possibly its transformation. The most creative second stage is based only on the declarative semantics of logic, and is independent of the target logic programming language used at the third stage.

In Chapter 5, we formulate the *objective* of this thesis in more detail, namely logic algorithm synthesis from specifications by examples and properties. A series of motivating sample scenarios allows us to identify the challenges of this objective.

Part II: *Building Blocks*

In Part II, we provide the *building blocks* that are used later for the development of a logic algorithm synthesis mechanism. These building blocks are a new language for expressing incomplete specifications (Chapter 6), a complete theoretical framework for the formulation of stepwise synthesis strategies (Chapter 7), an introduction to the notion of algorithm schemas and their usage in algorithm synthesis (Chapter 8), a method for deductively synthesizing parts of logic algorithms (Chapter 9), and another method for inductively synthesizing parts of logic algorithms (Chapter 10). Note that Chapters 7 to 10 may be read in any order.

In Chapter 6, we define a *specification approach* that is based on the notions of *examples* and *properties*. It requires examples that are chosen in a consistent way by a specifier who knows the intended relation. The presence of properties (whose actual language is application-specific, and thus left unspecified for a while) is meant to overcome the problems of ambiguity and limited expressive power of examples, while still preserving their virtues of naturalness and conciseness. Such specification languages are quite expressive and readable. Specifications by examples and properties are usually incomplete, and hence ambiguous, but minimal. This specification approach holds the promise of faster and more reliable synthesis than from examples alone.

In Chapter 7, we develop a complete *theoretical framework for stepwise synthesis of logic algorithms from specifications by examples and properties*. Three layers of new correctness criteria relating the intentions, specifications, and logic algorithms are introduced. Comparison criteria relating logic algorithms in terms of semantic or syntactic generality are then proposed. All these criteria provide an adequate structure for the formulation of stepwise synthesis strategies, be they incremental (examples and properties are presented one-by-one) or non-incremental (examples and properties are presented all-at-once). A particular non-incremental strategy is developed in greater detail for use in the sequel.

In Chapter 8, we discuss *algorithm schemas* as an important support for algorithm design. One of the major ideas of this thesis is that schema-independent methods can be developed for the synthesis of instances of the place-holders of schemas. We thus advocate a very disciplined approach to algorithm synthesis: rather than using a uniform method for instantiating all place-holders of a given schema (possibly without any awareness of such a schema), one should deploy the best-suited method for each place-holder. In other words, we propose to view research on automatic programming as: (1) the search for adequate schemas; (2) the development of useful methods of place-holder instantiation; and (3) the discovery of interesting mappings between these methods and the place-holders of these schemas. Our grand view of algorithm synthesis systems is thus one of a large workbench with a generic tool-box of specialized methods and a set of schemas that covers (as much as possible of) the space of all possible algorithms.

In Chapter 9, we develop the *Proofs-as-Programs Method*, which deductively adds atoms to a logic algorithm so that it satisfies a given set of properties. The added literals are extracted from the proof that the given algorithm is complete with respect to these properties. This method is part of our tool-box for instantiating place-holders of an algorithm schema.

In Chapter 10, we develop the *Most-Specific-Generalization Method*, which inductively synthesizes a logic algorithm from a set of examples. The intended relation, though unknown as a whole, is however known to feature a given data-flow pattern between its parameters. The synthesized logic algorithm is correct with respect to a "natural extension" of the given examples. Note that this method is also part of our tool-box for instantiating place-holders of an algorithm schema, but not a solution to the overall problem of synthesis from specifications by examples and properties.

Part III: *A Logic Algorithm Synthesis Mechanism*

In Part III, we develop an actual *logic algorithm synthesis mechanism* from specifications by examples and properties, as seen in Chapter 6. It fits the particular non-incremental synthesis strategy presented in Chapter 7, is guided by the divide-and-conquer algorithm schema seen in Chapter 8, and uses the tool-box of methods developed in Chapter 9 and Chapter 10.

In Chapter 11, we motivate the desired *features* of the mechanism (such as the actual language for the properties), and argue for a series of preliminary *restrictions*, so as to keep the presentation simple until the discussion of its extensions. We also give an intuitive *overview* of the entire synthesis mechanism by illustrating it on a sample execution, so as to give the reader the feel for its working.

In Chapter 12, we give full detail about the *expansion phase of synthesis*, that is the first four steps of the mechanism. These steps are rather straightforward, and do not require any sophisticated methods.

In Chapter 13, we give full detail about the *reduction phase of synthesis*, that is the remaining three steps of the mechanism. These steps are the truly creative ones, and require the sophisticated methods of the tool-box developed in Chapter 9 and Chapter 10.

In Chapter 14, we provide a detailed *evaluation* of the obtained synthesis mechanism with respect to the identified challenges. We also discuss some *extensions* to the synthesis mechanism. Then, we outline a *methodology for choosing "good" examples and properties*, which, when followed, increases reliability and speed of synthesis, and decreases the need for interaction with the specifier. A prototype *implementation* of this synthesis mechanism is being developed. It is called SYNAPSE (*SYNthesis of logic Algorithms from PropertieS and Examples*), and is written in portable Prolog. We briefly discuss its architecture, and list a few target scenarios of interaction between the specifier and the synthesizer. Finally, we give an *evaluation* of the power of the proposed synthesis mechanism, and compare it to related work.

I STATE OF THE ART

In this first part, the state of the art of the topics underlying this dissertation is presented, and a lot of relevant terminology is introduced along the way. The idea is to gradually introduce the objective of this research by narrowing in from more general aims. Thus, in Chapter 1, a general introduction to the field of automatic programming (program synthesis) is given. Chapter 2 contains a survey of the use of deductive inference in automatic programming, while Chapter 3 surveys the use of inductive inference in automatic programming. In Chapter 4, we summarize the methodology of systematic logic program development of [Deville 87, 90]. Chapters 2 to 4 may be read in any order. Finally, in Chapter 5, we formulate the objectives of this thesis.

1 Automatic Programming

There will certainly be many differences between the input to future automatic programming systems and what is currently called a program. However, programming is best typified not by what programs are but by what programming tasks are like. Undoubtedly these inputs will still have to be carefully crafted, debugged, and maintained according to changing needs. Whether or not one chooses to call these inputs programs, the tasks associated with them will be strongly reminiscent of programming.

— Charles Rich and Richard C. Waters.

In this chapter, a general introduction to the field of automatic programming (program synthesis) is given. In Section 1.1, the grand aim of automatic programming is presented, together with the major issues. Then, in Section 1.2, we discuss some topics around the languages used in program synthesis, while in Section 1.3, we classify the different synthesis mechanisms that have appeared so far. Finally, in Section 1.4, we outline our viewpoint on the requirements and promises of automatic programming.

1.1 The Grand Aim of Automatic Programming

Programming is hard! Indeed, programs are highly complex mathematical objects reflecting many inter-dependent decisions. For instance, in the imperative programming paradigm, initializing operations, loops and their invariants, and terminating operations have to be carefully crafted; convenient data structures have to be selected; operational correctness with respect to the specification and suitable time or space efficiency have to be achieved. It turns out that human beings are often inept at managing these difficulties. Even after intensive training and practice, errors such as one-too-many iteration still occur.

This contributes to the so-called *software crisis*, which has been haunting commercial software production for over two decades now. A common symptom is the inability to deliver correct programs on time and to budget. Growing needs for more and more complex software have made the crisis outpace the academic results for dealing with it.

Note that, at this level of the discourse, we are not interested in the languages used for writing specifications and programs. The main issue is that there should be criteria for assessing the correctness of a specification with respect to the informal intentions, and criteria for assessing the operational correctness of a program with respect to its specification. Sample criteria are partial correctness, completeness, and termination (the latter being meaningless for specifications).

Actually, a classification of problems is needed for further clarification. On the one hand, there are *data-processing problems*, where the specification is almost tantamount to an actual algorithm. On the other hand, there are *algorithmic problems*, where the specification is at best some kind of a naive (inefficient) program. Although this was implicit so far, the focus in this thesis is exclusively on the more challenging algorithmic problems. Moreover, this implies that we are here only concerned with programming-in-the-small.

Let's perceive, in a first approximation, the actual programming task of the software lifecycle as being divided into two stages:

(1) *elaboration* of a specification from informal intentions;
(2) *development* of a program that is operationally correct with respect to the specification.

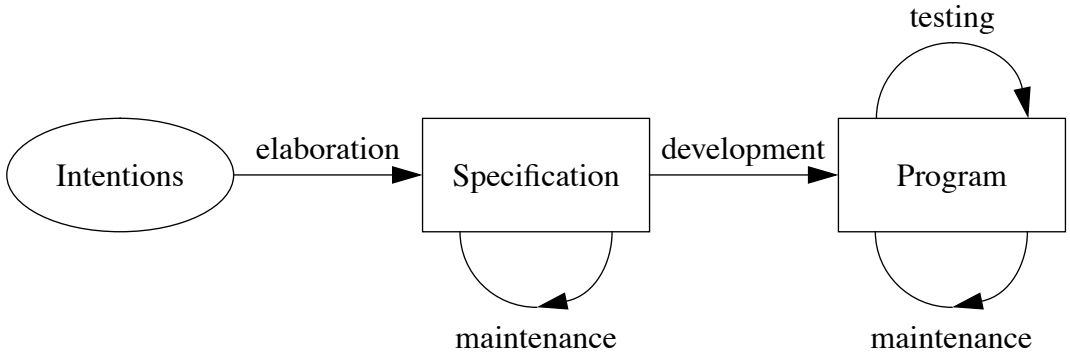


Figure 1-1: The traditional software life-cycle

However, this view assumes that stage (2) is "easy". In practice, the phrase "is correct" of that stage is often replaced by "seems correct", and a third stage is added:

(3) *testing* of the program to determine whether it is operationally correct with respect to the specification.

There are basically two approaches to testing. A first approach is *retrospective verification*, where one checks the correctness of a given program with respect to its specification. This turns out to be extremely hard by hand, not to mention automation. The other approach is *validation*, where the program is debugged so that it no longer fails on some test-data. However, and despite the still wide-spread usage of this approach, this is not satisfactory because debugging often amounts to patchwork, and because validation amounts to verification only in the limit.

Finally, specifications are sometimes perceived to be incorrect, and they need thus to be changed. And specifications sometimes evolve over time, as the intentions change. A fourth stage is then added:

(4) *maintenance* of the specification and of the program so that it is correct with respect to the new specification.

Figure 1-1 summarizes this perception of the programming task of the software life-cycle. However, maintenance sometimes operates directly on the old program, and is thus tantamount to a validation process. While justifiable for some minor changes to the specification, this approach is dangerous in general. A replay of stage (2) is definitely a better approach, even in case of inability to guarantee correctness.

So, how can the ideal formulation of stage (2) be achieved? A first solution is *constructive verification*, where the program is developed in a way that guarantees its correctness with respect to the specification. This makes sense as program development actually requires the same kind of formal thinking as program verification. The methodologies proposed by Dijkstra, Floyd, Hoare, Naur, Wirth,… since the late 1960s belong to this category. However, there was little concern about automating these methodologies.

The second solution is the dream of *automatic programming* (*program synthesis*), where some program develops correct programs from specifications, with as much (or as little) interaction as the specifier wants. This dream is as old as computer science itself, but often dismissed as the automa-g-ic programming dream. A more general approach than program verification is however to prove that some program synthesis mechanism is sound, and that it thus always synthesizes correct programs. The benefits of such a solution would be the disappearance of the program testing and program maintenance stages from the programming task, and instead a focus on specification elaboration, testing (via prototyping, paraphrasing, symbolic execution,…), and maintenance, because replay of program development becomes

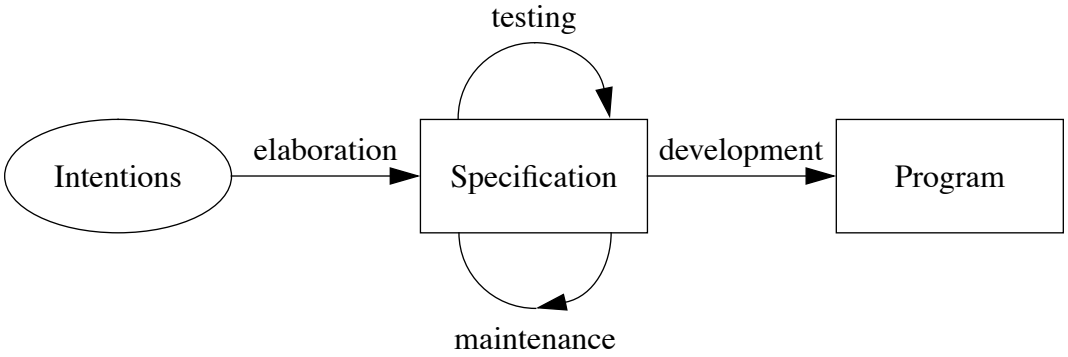


Figure 1-2: A new software life-cycle

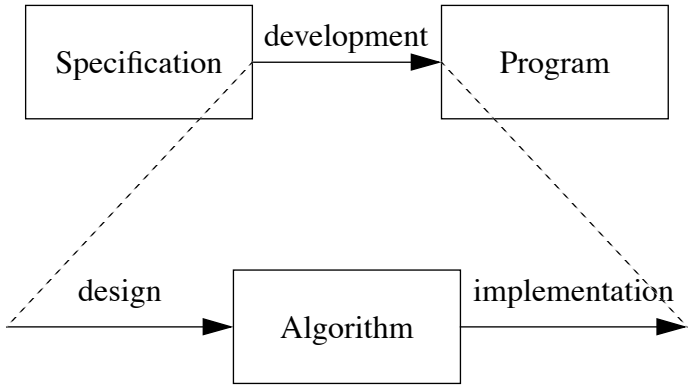


Figure 1-3: A practical view of program development

less costly. Figure 1-2 shows the resulting paradigm shift and its new perception of the programming task of the software life-cycle.

Let's for a while still keep from actually surveying the achievements of program synthesis research. Instead, we first aim at restricting the scope of such research. Indeed, many people believe it is beneficial to decompose the program development stage into two sub-stages (see Figure 1-3):

(1) *design* of an abstract algorithm, with prime concern about its logical correctness with respect to the specification, but not so much concern about its time or space efficiency;

(2) *implementation* of the algorithm as a concrete program written in some programming language, with concern about achieving operational correctness with respect to the specification and about maximizing time or space efficiency.

The advantages of this approach are twofold. First, the algorithm is not subjected to the computational model of the target programming language, nor to its deficiencies or even to its intended underlying machine architecture. Note that, at this level of the discourse, we are not interested in the language used for writing algorithms. The main issue is that there should be criteria for assessing the logical correctness of an algorithm with respect to its specification. Second, the implementation stage, now that it is clearly separated, may explicitly re-use the huge body of existing and ongoing research in algorithm transformation, algorithm implementation, and program transformation. The disadvantage of this approach is however that algorithm efficiency considerations cannot always be clearly dissociated from the algorithm design process: sometimes, at the transformation time, it is "too late" to improve an algo-

rithm. Special attention needs thus to be paid to make the synthesis of efficient algorithms possible, if not likely, rather than a mere pot-shot.

There is a growing awareness since the early 1980s (see [Kant 85] or [Bauer *et al.* 89], for instance) of the necessity to separate these concerns, but such hasn't always been the case. At best, some projects focussed on the so-called "pure" aspects of programming languages such as LISP and Prolog. In this thesis, except for the surveys in Chapter 2 and Chapter 3, we thus prefer algorithm design over program development. We also adopt the following terminology for distinguishing design approaches: *algorithm construction* stands for (completely) manual algorithm design (usually from informal specifications), whereas *algorithm synthesis* stands for (partly) automatic algorithm design (usually from formal specifications).

Even if the original scope of automatic programming is stripped down to sub-stage (1) only, we may still paraphrase our initial statement: Algorithm design is hard! Indeed, algorithms are highly complex mathematical objects reflecting many inter-dependent design decisions: initializing operations, loops and their invariants, and terminating operations have to be carefully crafted; logical correctness with respect to the specification has to be achieved. It turns out that human beings are often inept at managing these difficulties. Even after intensive training and practice, design errors such as one-too-many iteration still occur.

The root of the difficulty resides in the huge gap in content and formality between the intentions (informal description of what problem is to be solved) and the program (formal description of how that problem is solved). Even though algorithm synthesis research restricts its focus to bridging the smaller gap between the specification (description of what a program does) and the algorithm (description of how the program does it), this is still quite a formidable task.

1.2 Specification Languages, Algorithm Languages, and Programming Languages

In this section, we first briefly discuss issues related to the choice of specification languages, algorithm languages, and programming languages in automatic programming research. We then take a cynical look at the history of automatic programming, and draw some conclusions about formal languages.

Specification Languages

Ideally, a *specification* is only a description of what a program does, and an explanation of how to use the resulting program. A specification should faithfully capture the informal intentions, namely in that the problem at hand may be solved by the specified program.

Desirable qualities of a specification language are *expressiveness* (provision of constructs that are natural to human thinking within the targeted application domain) and *readability*.

A crucial aspect of a specification is its *total correctness* with respect to the informal intentions. Total correctness is often divided into *partial correctness* (absence of contradictions with the intentions) and *completeness* (absence of under-specification). While partial correctness of a specification with respect to the intentions is usually assumed (even if this is often far from obvious), such is not the case with completeness, and hence with total correctness.

Other desirable qualities of a specification are *minimality* (absence of over-specification), *internal consistency* (absence of internal contradictions), *non-ambiguity* (existence of a single interpretation), and *non-redundancy* (absence of synonymy, homonymy, and so on).

Sometimes, the notion of *expansion factor* (ratio between the size of an algorithm or program and the size of the corresponding specification) is proposed as a goodness measure for assessing a specification, or a specification language, or even a synthesis mechanism. This,

however, is a worthless measure, as everything depends on the specifier and on the verboseness and availability of libraries of the involved languages.

A specification language should offer abstract data-structures, so that specifications do not bias the design or implementation processes. However, abstract data-structures and the issue of their reification are not addressed anywhere in this thesis, and are considered future work.

Specifications may be written in a formal language or in a non-formal language. A *formal* specification language has a well-defined syntax and semantics. A *non-formal* specification language is not to be confused with pure natural language: just like proofs in mathematics textbooks, non-formal specifications use natural language constructs, but jointly with formal constructs that are introduced as the need arises. There are thus no predefined syntax and semantics. See [Le Charlier 85] for a complete treatment of this issue.

There seems to be no doubt that automated algorithm synthesis can only start from formal specifications, but we'll get back to this issue later in this section. A host of formal specification languages has been proposed, some especially for synthesis purposes. The best-known formal specification languages are very-high-level languages such as GIST, *Refine*, SETL, Z, and so on. Other formal specification languages are first-order logic (sometimes in the guise of frames or semantic nets), algebraic formalisms, examples, execution traces, state transition diagrams, and so on.

Programming Languages and Algorithm Languages

The most popular programming languages used as target languages of synthesis systems are, by far, LISP and Prolog. These languages are, by the way, also the most-used languages for implementing the synthesis mechanisms themselves, though Mathematica and expert system shells such as OPS-5 are also used. This is not surprising, due to the ease of reasoning about programs in these languages. However, due to the recent trend to synthesize algorithms rather than programs, the first aspect of this "domination" seems over, as the algorithm implementation stage may target virtually any programming language.

So what algorithm languages can possibly be used? The so-called pseudo-code languages used in many algorithms textbooks are certainly candidates, but they have received little attention so far, and probably won't ever. Indeed, the bulk of the algorithm language definition effort goes to the pure foundations of functional and logic programming, again because of the ease of reasoning about algorithms written in such languages.

A Cynical View on Formal Languages

Let's now lean back and take a look at the language aspect of the history of automatic programming. In computer science, description languages are traditionally classified as being more or less "high-level". At the "high" end of the continuum is natural language, as we use it everyday. At the "low" end of the continuum are the most basic machine languages, as they are actually directly interpreted by computers. Low-level languages are very informative in describing how a problem is actually solved, whereas high-level languages only allow a description of what the problem is all about.

Every new generation of programming languages makes a quantum leap towards the high end of the language continuum. For instance, assembly languages relieved programmers from tedious on/off switching. *Fortran* provided evaluation of formulas. *Algol* supplied abstractions of control structures. Structured programming languages such as *Pascal* brought some discipline into *Algol*-like programming, as well as abstractions of data-structures. Functional and logic programming languages such as LISP and *Prolog* introduced declarative programming. And so on. Simultaneously, formal specification languages have also been moving towards the high end of the language continuum.

So, whenever a program written in some new language looks like what is at that moment called a formal specification, and whenever compilation (or interpretation) of such a new programming language achieves what seems then akin to programming, there is a period of confusion about what formal specifications and programs actually are. One speaks of automatic programming for a while, at least until the new programming paradigm has soaked in. Program synthesis research is thus constantly trying to keep pace with formal specification languages (or, equivalently, to be one step ahead of the state-of-the-art in programming), but, in retrospect, it is nothing else but the quest for new programming paradigms. To paraphrase Tesler's theorem (though probably from a different mind-set): automatic programming is whatever hasn't been compiled yet! Of course, as our notion of programs evolves, our understanding of compilation has to evolve as well: it is not because today's compilers are largely deterministic and automatic that tomorrow's compilers (today's synthesizers) are not allowed to have search spaces or to be semi-automatic.

Now, considering that programming languages and formal specification languages are moving over time to the high end on the language continuum, it is easy to see that, in the limit, they will both converge to natural language. In other words, programming and specifying-in-a-formal-language are the same processes in the limit! The very fact that there are occasionally periods of confusion about what formal specifications and programs actually are, already hints at this. So programming actually amounts, in the limit, to a formalization process.

Most of the early program synthesis projects actually did aim at starting from natural language specifications. A survey on very early projects was made by [Heidorn 76]. The SAFE project [Balzer 85] at the Information Sciences Institute of the University of Southern California initially went to great efforts to start from natural language specifications [Balzer *et al*. 77], but then eventually switched to defining GIST, a very-high-level specification language. The PSI project at Stanford University [Green 77] included a strand of research on synthesis from natural language specifications. At MIT, the *Protosystem* I project [Ruth 78] aimed at generating programs for data-processing problems. At Duke University, there was the *Natural Language Computer* project [Biermann 83].

The aim of these projects is called *natural language programming*, and it still constitutes the ultimate goal of automatic programming research. Because of the limited success of these early projects, this strand of research seems more or less dormant nowadays.

1.3 A Classification of Synthesis Mechanisms

A huge variety of synthesis mechanisms have been developed, reflecting at least the variety of specification languages. These mechanisms could be classified according to whether they start from formal or informal specifications, or according to whether they start from complete or incomplete specifications, or according to whether they are completely automatic or only semi-automatic, or according to whether they perform *algorithmic* synthesis (no usage at all of heuristics) or *heuristic* synthesis (at least partial usage of heuristics). But such classifications tend to be very lopsided, and thus uninformative. Instead, we classify the mechanisms according to the predominant kind of reasoning:

- inductive synthesis:
  - trace-based synthesis;
  - model-based synthesis;
- deductive synthesis:
  - transformational synthesis;
  - proofs-as-programs synthesis;
  - schema-guided synthesis;
- abductive synthesis;

- analogical synthesis;
- knowledge-based synthesis:
  - empirical synthesis;
  - synthesis by inspection;
  - synthesis from natural language.

Actually, knowledge-based reasoning is not really a separate kind of reasoning to the same degree as inductive, deductive, abductive, or analogical inference. And synthesis mechanisms of the first four categories never perform their kind of inference in an entirely pure way, to the exclusion of all possible sources of knowledge. But it certainly helps to view knowledge-based synthesis, with a predominant usage of knowledge, as a separate category, even if other kinds of reasoning are used as well. Of course, this classification is not strict: many mechanisms would actually fit into more than one category.

A graph confronting the specification languages with the synthesis mechanisms would be very sparse, because some mechanisms are unique to specification languages. Achievements of inductive and deductive synthesis are surveyed in Chapter 2 and Chapter 3, respectively.

The history of research in automatic programming has pretty much followed the pattern set by problem solving research in artificial intelligence. Indeed, the original quest for a completely general, fully automatic, end-user-oriented automatic programming system has gradually been replaced by more realistic objectives. Following [Rich and Waters 88a], there are three major trends, each sacrificing one of these objectives in order to attain the other two:

- the *expert specifier* trend sacrifices end-user orientation in view of attaining domain-generality and full automation; in vogue until the late 1960s due to the good results in compiler design for high-level languages, this trend has reappeared in the early 1980s when the technology for handling very-high-level languages became available;
- the *narrow domain* trend sacrifices domain generality in view of attaining end-user orientation and full automation; this trend was mainly a fallback during the 1970s while progress in the expert specifier trend was stalled; however, this trend is slowly emerging again, as the idea of collections of narrow-domain synthesizers arises;
- the *software assistant* trend sacrifices full automation in view of attaining domain generality and end-user orientation; this trend has been very popular since the 1980s, although the focus is sometimes more on having a calculus of algorithm design, than on actually using it for synthesis.

Of course, these trends are not as clear-cut as their descriptions might suggest. Indeed, many systems sacrifice a little bit of every objective.

1.4 Requirements and Promises of Automatic Programming

In this last introductory section to automatic programming, we list a few requirements for research in program synthesis, and outline the promises of this research. Finally, pointers to introductory literature are given.

There first is a great need for incorporating knowledge into automatic programming systems, in order to overcome the old "sins" of general problem solving. There are essentially three classes of useful synthesis knowledge:

- *algorithms knowledge* is needed [Soloway and Ehrlich 84] for re-using human expertise in algorithm design, and thus for making synthesis a "disciplined" search process; such knowledge can be codified in various ways: as transformation rules (see the PECOS sub-system [Barstow 77, 79a, 79b, 84a] of PSI), as algorithm schemas (see the systems of [Smith 81, 85, 88]), as clichés (see the *Programmer's Apprentice* project at MIT [Rich 81] [Waters 85] [Rich and Waters 87, 88b, 89]), and so on;

- *domain knowledge* (*application knowledge*) is equally crucial, for the same reasons (see the PHINIX system of [Barstow 84b, 85]);
- *meta-knowledge* is knowledge about how and when to use the previous two knowledge sources (see the *Designer* project at CMU [Kant 85] [Steier and Kant 85]).

Furthermore, there is a need for *efficiency knowledge* (useful for algorithm transformation and implementation), but this carries us beyond our focus on actual synthesis. As usual, there is the dreaded knowledge acquisition bottleneck, but current machine learning techniques (see Chapter 3) seem a promising step towards overcoming this.

Next, there is a need for *evolutionary specifications*, where one deliberately omits some functionalities in a first specification, and then relies on re-play facilities [Wile 83] when adding the missing functionalities in subsequent versions of the specification. This form of exploratory programming promises an answer to the endemic difficulty of providing correct and complete formal specifications.

A synthesis system should be able to design a large family of algorithms from a single specification. The different design decisions taken at the choice-points during synthesis then document interesting algorithm classifications. A popular idea is to benchmark the ability of a synthesis system on the sorting problem, whose specification is deceptively simple, and yet gives rise to a tremendous variety of different algorithms. Classification through synthesis of sorting algorithms has indeed been done by [Green and Barstow 78], [Clark and Darlington 80], [Broy 83], [Follet 84], [Smith 85], [Lau 89], and [Lau and Prestwich 91]. Moreover, the ability to find new algorithms might be considered another benchmark of the ability of a program synthesis system.

Following [Kant 90], the "economics" of automated program synthesis are as follows. The potential *benefits* are an increased productivity of software engineers (focus on creative aspects, automation of tedious routine, replay facilities) and an increased quality of the developed software (correctness, efficiency, re-usability). The *costs* are the necessary acquisition of programming and domain knowledge, the development of the synthesis system and its maintenance across various platforms, and the training of expert users. From this cost-benefit analysis, it appears that the use of a program synthesis system is worthwhile if either many similar programs need to be written, or a given program is changed sufficiently often, and if the application domain lends itself to natural and concise specifications, features well-understood problem solving techniques, and is complex enough for promising a payoff. The *break-even point* is thus attained when the usage of a program synthesis system is more economical than manual program development by an expert.

Regarding the promises, there is general optimism that program synthesis systems will eventually scale up to realistic tasks, especially now that some systems are already in commercial or pre-commercial use. Due to its very focus on programming-in-the-small, it is obvious that automatic programming will not supplant the need for programming-in-the-large, nor will it assist in project management issues. Also, the essentially evolutive nature of software development cannot be prevented by program synthesis systems, though they will certainly help in coping with changing specifications. Programming jobs will not disappear, but our understanding of what the task of programming is will definitely change. There will always be a need for validation and maintenance, but the promise is to do this on texts that are at today's specification level.

Pointers to the Literature

A thought experiment by [Green *et al*. 83] has led to the definition of the ideal architecture and functionalities of the software engineering environment of the future, namely the so-called KBSA (*Knowledge-Based Software Assistant*). General introductions to program synthesis have been published by [Rich and Waters 86b, 88a] and [Kant 90].

A growing number of surveys of existing synthesis mechanisms and systems is being published: let's just mention those of [Barr and Feigenbaum 82], [Biermann *et al*. 84b], [Partsch and Steinbrüggen 83], [Smith 84], [Goldberg 86], [Feather 87], [Lowry and Duran 89], [Steier and Anderson 89], [Biermann 92], and [Deville and Lau 94].

Compilations of landmark papers on program synthesis have been edited by [Biermann and Guiho 83], [Biermann *et al*. 84a], [IEEE-TSE 85], [Rich and Waters 86a], and [Bibel and Biermann 93]. A journal entirely dedicated to program synthesis will (finally) exist in the near future: the first issue of the *Journal on Automated Software Engineering* (Kluwer Academic Publishers) will be released in early 1994.

Artificial intelligence conferences typically have sessions or workshops on program synthesis. Especially in the logic programming area, there are specialized publications, such as [Jacquet 93] and the proceedings of the LOPSTR (*LOgic Program Synthesis and TRansformation*) workshops [Clement and Lau 92] [Lau and Clement 93] [Deville 94].

2 Deductive Inference in Automatic Programming

Deductive inference is omnipresent in automatic programming. We first define, in Section 2.1, the underlying specification formalism, namely axiomatizations. In Section 2.2, we classify the various approaches to using deductive inference in automatic programming. Then, in Section 2.3 and Section 2.4, we survey synthesis of functional programs from axioms, and synthesis of logic programs from axioms, respectively. Finally, in Section 2.5, we draw some conclusions on the use of inductive inference in automatic programming.

2.1 Specifications by Axioms

We first define a possible language for specifications by axioms, and this in the logic programming framework. Later in this section, we list alternative approaches.

Definition 2-1: An *axiomatic specification* of a procedure for predicate *r*/*n* consists of the union of a set of first-order axioms about *r*/*n* and the axiomatic specifications of all the non-primitive predicates used in these axioms.

We consider at least =/2, ≠/2, and ≤/2 to be primitives. Others are added whenever the need for simplification arises. Let's illustrate this on a few sample specifications.

Example 2-1: The *member(E,L)* relation holds iff term *E* is an element of list *L*. Given a ground value of *L*, this is a non-deterministic relation. A possible axiomatic specification is:

```
∀E ∀L member(E,L) ⇔
                append(_,[E|_],L)
```

with *append*/3 considered to be a primitive that holds iff its third parameter is the concatenation of its first two parameters, which must be lists. ♦

For the next two sample specifications, we need two new notions. Informally speaking, a *plateau* is a non-empty list of identical elements. A *compact list* is a list of couples, where the first term of a couple, called the *value* of the couple, is different in two consecutive couples, and the second term of a couple, called the *counter* of the couple, is a positive integer.

Example 2-2: The *firstPlateau(L,P,S)* relation holds iff *P* is the maximal plateau at the beginning of the non-empty list *L*, and list *S* is the corresponding suffix of *L*. Given a ground value of *L*, this is a fully deterministic relation. A possible axiomatic specification is:

```
∀L ∀P ∀S firstPlateau(L,P,S) ⇔
                plateau(P) ∧ ¬sameFirst(P,S) ∧ append(P,S,L)
∀P plateau(P) ⇔ ∃U ∀E
                member(E,P) ⇒ E=U
∀P ∀L sameFirst(P,L) ⇔ ∃E ∃F
                P=[E|_] ∧ L=[F|_] ∧ E=F
```

with *append*/3 and *member*/2 as in the previous example. ♦

Example 2-3: The *compress(L,C)* relation holds iff *C* is a compact list of $<v_i,c_i>$ couples, such that the i^{th} plateau of list *L* has c_i elements equal to v_i. Given a ground value of either parameter, this is a fully deterministic relation. A possible axiomatic specification is:

```
∀L ∀C compress(L,C) ⇔ ∃E ∃P ∃S ∃T ∃N
    L=[]        ∧ C=[]
  ∨ L=[E|_]     ∧ firstPlateau(L,P,S)
                ∧ compress(S,T)
                ∧ length(P,N)
                ∧ C=[<E,N>|T]
```

```
∀L ∀N length(L,N) ⇔ ∃T ∃M
    L=[]       ∧ N=0
  ∨ L=[_|T]    ∧ length(T,M)
               ∧ N=s(M)
```

with *firstPlateau*/3 as in the previous example. ♦

Example 2-4: The *sum(L,S)* relation holds iff *S* is the sum of the elements of integer list *L*. Given a ground value of *L*, this is a fully deterministic relation. A possible axiomatic specification is:

```
∀L ∀S sum(L,S) ⇔ ∃H ∃T ∃N
    L=[]       ∧ S=0
  ∨ L=[H|T]    ∧ sum(T,N)
               ∧ add(H,N,S)
```

with *add*/3 considered to be a primitive that holds iff its third parameter is the sum of the first two parameters, which must be integers. ♦

Example 2-5: The *sort(L,S)* relation holds iff *S* is an ascendingly ordered permutation of integer list *L*. Given a ground value of *L*, this is a fully deterministic relation. A possible axiomatic specification is:

```
∀L ∀S sort(L,S) ⇔
    permutation(L,S) ∧ ordered(S)
∀L ∀P permutation(L,P) ⇔ ∀U
    member(U,L) ⇒ ∃N count(L,U,N) ∧ count(P,U,N)
∀L ordered(L) ⇔ ∀U ∀V ∀A ∀B
    append(A,B,L) ∧ member(U,A) ∧ member(V,B) ⇒ U≤V
∀L ∀E ∀N count(L,E,N) ⇔ ∃H ∃T ∃M
    L=[]       ∧ N=0
  ∨ L=[E|T]    ∧ count(T,E,M)
               ∧ N=s(M)
  ∨ L=[H|T]    ∧ H≠E
               ∧ count(T,E,M)
```

with *append*/3 and *member*/2 as in the previous examples. ♦

It should be noted that the sample specifications above do not reflect the only possible way of expressing axioms. Specification approaches vary according to the following criteria:

- *language*: axioms may be given as statements in (some subset of) first-order logic (with or without equality), or as algebraic statements, and so on;
- *directionality*: axioms may be in relational form (where there is no bias about whether parameters are input parameters or output parameters), or in functional form; for instance, in the case of logic as specification language, relational axioms take form (1) below, whereas functional axioms take form (2) hereafter:

$$\forall \mathit{Parameters} \quad \mathit{Pre(Parameters)} \Rightarrow (\mathit{pred(Parameters)} \Leftrightarrow \mathit{Post(Parameters)}) \qquad (1)$$

$$\forall \mathit{Inputs}\ \exists \mathit{Outputs} \quad \mathit{Pre(Inputs)} \Rightarrow \mathit{Post(Inputs, Outputs)} \qquad (2)$$

  where *Pre* is an optional pre-condition (constraints on the parameters), *Post* is a post-condition (relation between the parameters), and *pred* is the specified predicate;
- *recursion*: axioms are sometimes constrained to be non-recursive, but most often, recursion is allowed in axioms;
- *connectives*: axioms are sometimes constrained to be implication statements, or equivalence statements; sometimes, even both kinds of statements are allowed.

The specifier is usually assumed to be a human being, who is in turn assumed to know—even if only informally—the functionality of the intended program. Moreover, it is usually assumed that the given axioms are consistent with these intentions.

Anyway, whatever the actual setting, the major potential advantage of axiomatic specifications is the following:

+ *completeness* and *non-ambiguity*: a correct axiomatic specification completely and unambiguously defines the intended relation, which is thus equal to the specified relation;

The disadvantages of axiomatic specifications are:

– *artificiality*: human beings rarely resort to axioms in order to explain some concept; moreover, reading such axioms is very difficult, even to experts; finally, elaborating such axioms is also tough, and potentially as error-prone as directly writing the desired program; especially the satisfaction of the completeness requirement is very hard to ensure;
– *length*: axioms are a quite lengthy way of intensionally specifying some concept, even if a large set of powerful primitives is available, because the axiomatic specifications of these primitives are technically part of the specification of the top-level predicate.

Approaches for tackling these drawbacks include evolutionary specifications, where one deliberately omits some functionalities in a first specification, and then relies on re-play facilities when adding these functionalities in subsequent specifications.

Since the inception of logic programming, the line between logic specifications and programs has somewhat disappeared, if one considers that specifications ought to be formal, that is. See [Le Charlier 85] for an argumentation that specifications should be non-formal. We also adhere to that point of view. The placement of this line is historically controversial anyway.

For instance, the sample specifications above of *compress*/2, *length*/2, *sum*/2, and *count*/3 are actually already some kind of first-order logic programs, and are easily transformable into, say, Prolog programs. Indeed, these specifications are recursive and exhibit induction over some parameter; avoiding this leads to clumsy specifications. These specifications thus seem to bias the actual programming process because they already show a way of how to solve the problem, rather than stating what the problem is.

Sometimes, the line between "what" and "how" is non-existent: for instance, the specification of *sameFirst*/2 is non-recursive, but it nevertheless also incorporates the way of how to solve the problem.

The other sample specifications above seem more descriptive, because they are non-recursive, and thus only seem to state what the problem is. We write "seem" twice in the previous sentence, because one tends to look at such specifications through glasses tainted by the current state-of-the-art of (logic) programming. Indeed, it is not because we don't (yet) consider implications and universal quantifiers in the right-hand sides of equivalence statements to have some computational contents that they do not do so. In a sense, even such specifications are actually logic programs, though they (usually) embody very inefficient ways of how to solve the problems.

The objective of deductive synthesis research is to find mechanisms of translating such possibly inefficient "programs-of-tomorrow" into efficient "programs-of-today", and this unbiased by any possibly existing recursion in the specification. This amounts to changing our current understanding of what formal specifications and programs are, and to viewing synthesis as a translation (compilation, transformation) process.

2.2 Deductive Inference

Deductive inference is the process of obtaining new sentences (called *theorems*) from a set (called *theory*) of given sentences (called *axioms*), by application of rules of deductive inference. Theorems are also called logical consequences of the theory. We assume that the reader is familiar with deductive inference, and thus just repeat the basic definition.

Definition 2-2: Given a theory $\mathcal{A}$, a sentence G is a *logical consequence* of $\mathcal{A}$ if every model of $\mathcal{A}$ is a model of G. This is denoted $\mathcal{A} \models G$.

Note that deductive inference is always sound. Typical rules of deductive inference are *modus ponens*, *universal instantiation*, *resolution*, *mathematical induction*, and so on. The branch of artificial intelligence research that is bent on automating deductive inference is called *automated theorem proving*.

Again, since deductive inference is generally a familiar notion, we do not actually survey it. We rather directly propose a taxonomy of the different ways of applying deductive inference to program synthesis from axiomatic specifications. Thus, Section 2.2.1 to Section 2.2.3 respectively contain general introductions to transformational synthesis, proofs-as-programs synthesis, and schema-guided synthesis.

2.2.1 Transformational Synthesis

In transformational synthesis, a sequence of meaning-preserving transformation rules is applied directly to a specification, until a program is obtained. This kind of stepwise forward reasoning is akin to constructing a derivation tree, and is feasible with axiomatic specifications that link their parameters in a relational way:

$$\forall \mathit{Parameters} \quad \mathit{Pre(Parameters)} \Rightarrow (\ \mathit{pred(Parameters)} \Leftrightarrow \mathit{Post(Parameters)} \ )$$

where *Pre* is an optional pre-condition (constraints on the parameters), *Post* is a post-condition (the specified relation between the parameters), and *pred* is the specified predicate.

Applicability conditions are often attached to the transformation rules. Transformational synthesis is obviously an outgrowth of program transformation (optimization) research. And, as hinted in the previous section, there is only a fine line—if any—between program synthesis and program transformation.

There are *atomic transformation rules*, such as *unfolding* (which mimics the execution mechanism of the target language), *folding* (which performs the reverse transformation of unfolding), *universal instantiation*, *abstraction, predicate definition*, and various (possibly conditional) rewrite rules for the target language and lemmas of the application domain.

The objective of applying transformations is to filter out a re-formulation of the specification, so that recursion (or a loop) may be introduced by a folding step. This usually involves a sequence of unfolding steps, then some rewriting, and finally the folding step.

These atomic transformation rules constitute a sound and complete set for exploring the search space. However, they lead to very tedious and lengthy syntheses. They are myopic in the sense that there is no real plan about where to go, except for the above-mentioned objective of introducing recursion. The "eureka" about when and how to define a new predicate is difficult to find automatically. It is also hard to decide when to stop unfolding. There is a need for loop-detection techniques to avoid infinite synthesis through symmetric transformations.

The idea for overcoming these drawbacks is of course to define *macroscopic transformation rules* that are higher-level in the sense that they are closer to actual programming decisions, such as adding an accumulator parameter. Such macroscopic rules could be inferred by learning from sample syntheses based on the atomic rules. The major problem is of course to provide such a set of macroscopic rules that is still sound and complete. In view of facili-

tating the understanding of a synthesis, and allowing its replay in case a specification changes, the usage of transformation rules should be recorded.

Note that transformational synthesis (and program transformation) is closely related to EBL (*Explanation-Based Learning*), an analytic branch of machine learning (see Chapter 3). Indeed, the sequence of transformations from a specification (seen as a non-operational description) to a program (seen as an operational description) amounts to an explanation of the specification. Cross-fertilization between these two areas has led to EBPT (*Explanation-Based Program Transformation*) [Bruynooghe and De Schreye 89], where sample concrete transformations are used to guide the overall abstract transformation process.

2.2.2 Proofs-as-Programs Synthesis

The proofs-as-programs approach to program synthesis—also classified as *constructive synthesis*—is based on the Curry-Howard isomorphism [Howard 80], which states that there is a one-to-one relationship between a (constructive) proof of an existence theorem and a program that actually computes witnesses of the existentially quantified variables of the theorem. Given a specification in the functional form:

$$\forall \mathit{Inputs}\ \exists \mathit{Outputs} \quad \mathit{Pre(Inputs)} \Rightarrow \mathit{Post(Inputs, Outputs)}$$

the idea is to proceed in two steps:

(1) (constructively) prove the satisfiability of a statement expressing that a realization of this specification exists;

(2) obtain the method, embodied in the proof, of realizing the specification.

For the second step, one distinguishes between two approaches:

- the *interpretative approach* directly interprets the proof as a program, namely by means of an operational semantics defined on proofs;
- the *extractive approach* mechanically extracts ("compiles") a program, in a given target language, from the proof.

Both approaches have complementary advantages and drawbacks: interpretation is not as efficient as execution of a compiled version, but the choice of a target language might obscure computational properties of proofs.

In any case, the crux is of course the state-of-the-art in theorem proving: the key issues are soundness of the synthesis (entailing correctness, completeness, and possibly termination of the resulting programs), deductive power (provability for an entire class of statements, extractability of whole families of programs for the same specification), and efficiency (need for proof planning in order to control the huge search space). Traditional theorem provers, such as the one of [Boyer and Moore 79], are inadequate due to their inability to reason constructively about existential quantifiers. A common grudge is that the used specification form naturally leads only to the synthesis of total functions, but not of full-fledged relations (partial, multi-valued, possibly non-terminating predicates, that is). Solutions to this problem are currently being worked out (see Section 2.4.2).

The idea of exploiting (constructive) proofs as programs is actually way older than its naming as the Curry-Howard isomorphism in 1980: the idea is inherent to intuitionistic logic (see the work of Kleene in the 1940s), and the oldest synthesis systems that we could definitely identify as being part of this paradigm are those developed in the late 1960s by [Green 69] and [Waldinger and Lee 69]. The terminology itself seems to have been coined by Robert Constable in the early 1970s [Bates and Constable 85].

There is an obvious interest in proofs-by-induction, because these allow the synthesis of recursive programs. Note that there often is a similar "proof" content in transformational synthesis and proofs-as-programs synthesis, and it seems that the same proof construction tech-

niques should be applicable to both. This may suggest that these approaches are probably two facets of the same process. For instance, the work of [Neugebauer 92] shows how the specification forms of these approaches may be reconciled.

Just as there is a need for higher-level transformation rules in transformational synthesis, there here is a need for proof planning in order to incorporate expert knowledge. This meta-synthesis issue is usually tackled by the use of tactics. Following [Bundy 88], a *tactic* is a meta-program within a theorem prover that guides the application of the inference rules. For instance, a tactic could set up a proof-by-induction. Tactics should operate at the level of proof ideas such as those that are typically used by human provers. For synthesis, there should be tactics that embody actual programming knowledge (design strategies, efficiency considerations, and so on), and not just tactics that embody proof knowledge. Since tactics are programs, they can probably also be synthesized, and the correctness of tactics must be proven. In view of facilitating the understanding of a proof, and allowing its replay in case a specification changes, the usage of tactics should be recorded. Tactics may be bundled together to form *strategies*. A *proof method* is then a meta-level specification of a tactic, and a *proof plan* is a meta-level specification of a strategy.

Compared to transformational synthesis, there is no problem here about when to stop: synthesis halts when the proof is completed. Transformational synthesis seems more appropriate for synthesis from specifications that are almost programs (in which case synthesis is more like an optimizing transformation), whereas proofs-as-programs synthesis seems more appropriate for synthesis from highly descriptive specifications. Note that program transformation may actually be performed by the transformation of synthesis proofs.

However, there seems [Sintzoff 93] to be some pessimism of late regarding the wellfoundedness of pursuing this strand of research: it appears indeed that obtaining constructive proofs is an order of magnitude harder than writing programs. It would thus be preferable to write programs directly, rather than going through the tedium of proving their specifications first and then extracting programs therefrom. Moreover, if the conjectured isomorphism between the proofs-as-programs approach and the transformational approach holds, then the latter seems "doomed" as well.

2.2.3 Schema-Guided Synthesis

Programs can be classified according to their design strategies, such as divide-and-conquer, generate-and-test, top-down decomposition, and so on. Informally, a *program schema* is a template program with a fixed control flow, but without specific indications about the parameters or the actual computations. A program schema thus represents a whole family of particular programs that can be obtained by instantiating the place-holders to particular parameters or code. It is therefore interesting to guide program design by a schema that captures the essence of some strategy. Schemas are discussed in more detail in Chapter 8.

Schema-guided synthesis is likely to involve deductive inference, hence its classification in this category. Moreover, it is actually an answer to the complexity and size problems of both proofs-as-programs synthesis and transformational synthesis. Indeed, the application of a schema may be seen as the application of a macroscopic transformation rule that embodies very-high-level design decisions. And the application of a schema may also be seen as a tactic that embodies very-high-level design knowledge, rather than proof knowledge. However, we consider that schema-guided synthesis deserves its own sub-category, especially because, as hinted above, the previous two sub-categories might be similar anyway.

2.3 Functional Program Synthesis from Axioms

The synthesis of functional programs (LISP programs, say) has received wide-spread attention since the late 1960s, due to the ease of reasoning about functional programs. The surveys of [Barr and Feigenbaum 82], [Biermann *et al*. 84b], [Partsch and Steinbrüggen 83], [Goldberg 86], [Feather 87], [Lowry and Duran 89], [Steier and Anderson 89], and [Biermann 92] include discussions of the landmark systems. Similarly for the compilations edited by [Biermann and Guiho 83], [Biermann *et al*. 84a], [IEEE-TSE 85], [Rich and Waters 86a], and [Bibel and Biermann 93].

This present survey is organized as follows: Section 2.3.1 to Section 2.3.3 respectively survey the achievements of transformational synthesis, proofs-as-programs synthesis, and schema-guided synthesis of functional programs from axiomatic specifications.

2.3.1 Transformational Synthesis of LISP Programs

The pioneering work in LISP program transformation is the research of [Burstall and Darlington 77], who present a strategy and a semi-automated system for transforming (optimizing) recursive equations into tail-recursive ones, using folding, unfolding, instantiation, abstraction, and eureka-guided predicate definitions. The developed system was also used for LISP program synthesis [Darlington 81, 83].

The same results, though directly aimed at program synthesis, were obtained simultaneously, but independently, by [Manna and Waldinger 79]. However, their DEDALUS system (the *DEDuctive ALgorithm Ur-Synthesizer*, which is the successor of SYNSYS [Manna and Waldinger 77]) is automated. Moreover, particular care is taken to check input and termination conditions before introducing recursion by folding. Heuristics are used, but at the expense of possibly missing a better algorithm.

The field of *transformational implementation* is about synthesis of programs from specifications in very-high-level languages. This requires very large sets of transformation rules that redefine the very-high-level constructs of the specification language in terms of the programming language.

The 15-year SAFE project [Balzer 85] at the Information Sciences Institute of the University of Southern California aimed at transformational synthesis from specifications written in the GIST language. Some of the used transformation rules are explained by [London and Feather 82]. Ways of mechanizing the application of transformation rules are embodied by the GLITTER sub-system of [Fickas 85].

The PSI project at Stanford University [Green 77] [Green *et al*. 79] was built around such a transformational engine, called PECOS [Barstow 77, 79a, 79b, 84a]. The incorporated programming knowledge is further discussed by [Green and Barstow 78]. Tight heuristic-based interaction [Kant and Barstow 81] of PECOS with the efficiency expert LIBRA [Kant 77, 83] ensures efficient synthesis of efficient programs. A successor system, called CHI [Smith *et al*. 85], was partly developed at Kestrel Institute. The system was shown to be able to synthesize its own rule-compiler [Green and Westfold 82].

The role of domain knowledge codified as transformation rules has been advocated in the context of the PHINIX system [Barstow 84b, 85] at Schlumberger-Doll Research.

2.3.2 Proofs-as-Programs Synthesis of LISP Programs

As mentioned above, the probably first proofs-as-programs synthesis systems are QA3 (*Question-Answering system*) of [Green 69], and the *PROgram Writer* (PROW) of [Waldinger and Lee 69]. In the latter, a post-proof processor extracts LISP programs from constructive existence proofs.

The theoretical foundations have then been laid out by numerous people, such as [Martin-Löf 79], [de Bruijn 80], [Howard 80], [Beeson 85], [Constable *et al*. 86], [Coquand and Huet 86], [Hayashi 86], and so on, the last three also proposing implementations of actual program synthesis systems, such as the one of [Mohring 86], which is based on Coquand and Huet's calculus of construction. More recent achievements can be found in [Nordström *et al*. 90] and in [Huet and Plotkin 91].

The DEDALUS mechanism (see the previous section) has been rephrased in the proofs-as-programs paradigm [Manna and Waldinger 80, 91]. The so-called *deductive tableau method* is used for the proofs. There is no loss of synthesis power, but the mechanism is expressed in a considerably more elegant manner. An interesting insight is that constructive logics are not necessarily required for proofs-as-programs synthesis: indeed, many derivation steps during synthesis actually are only verification steps, and thus need thus not be constructive at all. Classical logic seems thus sufficient, provided it is "sufficiently" constructive when needed. The main focus seems to be on having a calculus of program synthesis, showing the derivability of many (hopefully all, that is) algorithms, including very intricate ones such as unification [Manna and Waldinger 81], and interesting new ones that haven't been hand-constructed yet [Manna and Waldinger 87]. But there is no mention yet of proof planning.

2.3.3 Schema-Guided Synthesis of LISP Programs

The use of deduction in the schema-guided synthesis paradigm is pretty much the brainchild of Douglas R. Smith. A first system was called NAPS (*Navy Automatic Programming System*) [Smith 81]. A technique for the derivation of pre-conditions (which is a generalization of theorem proving and of formula simplification) constitutes the deductive corner-stone of this system and of all its successors [Smith 82]. Indeed, schema-guided synthesis often requires deriving pre-conditions of a formula constructed by instantiation of a schema with information extracted from the specification and the partially designed algorithm. Such pre-conditions are used to either strengthen or instantiate some part of the algorithm.

Next came the *Cypress* system [Smith 85], which is based on a divide-and-conquer schema. Specifications of sub-problems are derived from the adaptation of a schema to the top-level specification, and recursively so on, until primitive problems are reached. This top-down problem reduction is then followed by a bottom-up composition of the synthesized sub-algorithms into the top-level algorithm, according to the schema. The system is interactive, yields totally correct algorithms, and is even able to cope with partial specifications.

The successor system, called KIDS (*Kestrel Interactive Development System*) [Smith 88, 90], handles schemas for design strategies such as divide-and-conquer, local search, and global-search (an enumerative search strategy that generalizes many known search strategies, such as binary search, backtracking, branch-and-bound, constraint satisfaction, and so on). Much effort has been put into transformation techniques for optimizing the synthesized algorithms: this subsequent transformation process gives this system the flavor of transformational synthesis. KIDS is believed to be very close to the break-even point where its usage is more economical than manual algorithm construction by an expert.

2.4 Logic Program Synthesis from Axioms

The synthesis of logic programs (Prolog programs, say) is an area of intense research since the late 1970s. Since the synthesized program is a logic program, the axioms are presented in the relational from. A detailed survey is being prepared by [Deville and Lau 94], and compilations of papers are available [Jacquet 93]. Specialized workshops are held annually within the LOPSTR (*LOgic Program Synthesis and TRansformation*) series [Clement and

Lau 92] [Lau and Clement 93] [Deville 94]. More general meeting grounds are the biannual META (*Meta-programming in logic programming*) workshops, and (workshops at) the various logic programming conferences. Correctness issues of logic programming are addressed by [Clark and Tärnlund 77], [Clark 79], [Hogger 81, 84], and [Deville 90], among others. This present survey is organized as follows: Section 2.4.1 to Section 2.4.3 respectively survey the achievements of transformational synthesis, proofs-as-programs synthesis, and schema-guided synthesis of logic programs from axiomatic specifications.

2.4.1 Transformational Synthesis of Prolog Programs

In order to give a rough feel for transformational synthesis of logic programs, we first present the pioneering mechanism in some detail. Then, we cite the achievements of some of the more recent mechanisms.

Synthesis by Symbolic Execution

Some of the early efforts were conducted at Imperial College of Science and Technology in London (UK). The developed synthesis mechanisms, called "*symbolic execution*", were inspired by the foundational work on LISP program transformation of [Burstall and Darlington 77].

Under the first approach [Clark and Sickel 77] [Clark 79, 81], the specification is "executed" with symbolic values that cover all possible forms of the type of some chosen induction parameter. For instance, if that parameter is a list, then the specification is "executed" with the symbolic values [] and [*H*|*T*]. "Execution" is a transformational process, based on a predefined set of transformation rules, whose aim it is to obtain a re-formulation of the specification that is (1) recursive, and (2) under the form of a definite Horn clause procedure. The transformation rules include *folding* (replacing a sub-formula by an atom), *unfolding* (replacing an atom by its definition; this mimics the execution mechanism of Prolog), logical rewrite and simplification rules, domain-specific rewrite and simplification rules, *specification introduction*, and many others. Recursion is thus obtained by folding a variant of the body of the specifying axiom into a recursive call.

A similar approach was taken by [Hogger 78, 81, 84], though with slight differences. Unfolding is referred to as "*resolution*", which is perfectly valid and underlines the operational flavor of "symbolic execution". More importantly, the induction on some parameter is only introduced as the need arises, and this by strengthening an implicant with a form-identifying atom, such as *L*=[*H*|*T*].

We now illustrate all this on a sample synthesis.

Example 2-6: The synthesis of a logic program for *member*/2 goes as follows. As a reminder, the top-level specification is (for convenience, we omit quantifiers):

$$member(E,L) \Leftrightarrow append(P,[E|S],L) \qquad (1)$$

Suppose that *append*/3 is defined as follows:

```
append(A,B,C) ⇔
    A=[]        ∧ C=B
  ∨ A=[H|T]     ∧ append(T,B,V)
                ∧ C=[H|V]
```

Suppose now that parameter *L* is chosen as the induction parameter. It is a list, so let's perform a symbolic execution for the mutually exclusive symbolic values [] and [*H*|*T*]. Let's start with the base case, and apply the substitution {*L*/[]} to (1):

$$member(E,[]) \Leftrightarrow append(P,[E|S],[])$$

Unfolding the *append*/3 atom yields, after some logical simplifications:

$$member(E,[]) \Leftrightarrow (false) \vee (false) \quad (2)$$

Let's now pursue with the structure case, and apply the substitution $\{L/[H|T]\}$ to (1):

$$member(E,[H|T]) \Leftrightarrow append(P,[E|S],[H|T])$$

Unfolding the *append*/3 atom yields:

$$member(E,[H|T]) \Leftrightarrow (P=[] \wedge [H|T]=[E|S]) \vee (P=[Q|U] \wedge append(U,[E|S],V) \wedge [H|T]=[Q|V])$$

Folding the *append*/3 atom according to (1) introduces the wanted recursion:

$$member(E,[H|T]) \Leftrightarrow (H=E) \vee (P=[Q|U] \wedge member(E,V) \wedge [H|T]=[Q|V])$$

Some logical simplifications yield:

$$member(E,[H|T]) \Leftrightarrow H=E \vee member(E,T) \quad (3)$$

The obtained statements (2) and (3) are easily transformed into definite clause form:

```
member(E,[H|T]) ← H=E
member(E,[H|T]) ← member(E,T)
```

This completes the synthesis. It is admittedly a little bit contrived because its linearity skips the issues of transformation rule selection and backtracking, but the point is here to get a feel for this kind of synthesis mechanism. ♦

Synthesis by symbolic execution obviously features the usual advantages and drawbacks of transformational synthesis.

Other Approaches

Folding/unfolding transformations were the object of a first formal study in logic programming by [Tamaki and Sato 84], although these transformations were already present in the above-mentioned symbolic execution synthesis approaches.

A highly structured top-down strategy for applying folding and unfolding, guided by a recursion schema provided by the specifier, is proposed by [Lau 89] [Lau and Prestwich 90, 91]. A semi-automated system was developed for assisting such syntheses.

The research of [Kraan *et al*. 92] takes a novel approach: a logic program is synthesized as a by-product of the planning of a verification proof of the specification. This proof is performed, at the object level, in a sorted, first-order logic with equality. The proof is planned, at the meta-level, while initially having the actual body of the extracted program represented by a second-order variable (this is called *middle-out reasoning*). As the planning proceeds by applying tactics to a conjecture (which is tantamount to applying transformation rules), the program becomes gradually instantiated. This requires an extension of the used *Clam* proof planner (see Section 2.4.2). This technique may obviously also be classified into the proofs-as-programs paradigm.

The LOPS (*LOgical Program Synthesis*) system of [Bibel 80] [Bibel and Hörnig 84] [Neugebauer 92], though originally described within the proofs-as-programs synthesis paradigm, actually performs transformational synthesis. A very small set of hardwired heuristics (which could be seen as design tactics) leads to the uniform synthesis of a recursive re-formulation of the specification. Another main originality of this system is that, guided by the syntax of the specification, it tries to discover itself the crucial knowledge about how to solve the base case of a proof-by-induction, and about how to compose partial results in the structure case of such a proof.

Several researchers have tried to make synthesis a deterministic process, akin to compilation. For instance, implication formulas with arbitrary bodies may be normalized into Horn clauses with negation by the so-called *Lloyd-Topor translation process* (see [Lloyd 87]). This is, however, a non-deterministic translation. Moreover, it does not always yield executable logic programs, due to the deficiencies of SLDNF resolution, such as floundering. Sometimes, the obtained programs are hopelessly inefficient. The work of [Sato and Tamaki 84] shows how to combine the folding/unfolding rules [Tamaki and Sato 84] with their so-called *negation technique* in order to perform an almost deterministic synthesis of definite logic programs from non-recursive specifications. However, the search space is way too large due to the generate-and-test flavor of the mechanism. The same team eventually did come up with a completely deterministic and automatic synthesizer [Sato and Tamaki 89], which is based this time on the notion of universal continuation forms, as well as the folding/unfolding rules. This "compiler" starts from a specification written as a set of clauses whose bodies are conjunctions of atoms or universally quantified implications. Either it finitely produces a definite clause program that is guaranteed to be partially correct wrt the specification, or it aborts for lack of logical power.

The work of [Waldau 91] is a first step towards proving the soundness of transformation rules for logic programs.

2.4.2 Proofs-as-Programs Synthesis of Prolog Programs

Various systems exist for doing constructive synthesis of logic programs. We simply enumerate them here, but give no insights into their underlying mechanisms, nor do we illustrate them on sample syntheses.

First, a large-scale effort was conducted by the UPMAIL group at the University of Uppsala (Sweden). The development of a logic programming calculus [Clark and Tärnlund 77] [Tärnlund 78, 81] [Hansson and Tärnlund 79] [Hansson *et al*. 82], which is based on Prawitz' natural deduction system for intuitionistic logic, led to a nice unified framework for logic program synthesis, verification, transformation, and execution. [Hansson 80] shows how to extract logic programs from constructive proofs-by-induction performed within this system. [Eriksson 84] used this mechanism for one of the first syntheses of a unification algorithm, which is considered one of the ultimate benchmarks of automatic programming. A proof editor, *NatDed* [Eriksson and Johansson 81, 82] [Eriksson *et al*. 83], was developed for assisting in a proof process. This editor was subsequently optimized to successfully recognize symmetry [Johansson 84, 85] and transitivity in proofs.

Then, there is *Nuprl* [Constable *et al*. 86], a proof development system that is based on the intuitionistic second-order type theory of [Martin-Löf 79], and that has been used for the synthesis of deterministic logic programs by [Bundy 88]. A first-order subset of the *Oyster* proof development system—a re-implementation of *Nuprl* in Prolog—was used for logic program synthesis by [Bundy *et al*. 90], with special focus on the synthesis of logic programs that compute full-fledged relations, and not just total functions. In view of automating proofs, a proof-planner called *Clam* was adjoined to *Oyster*. That overall effort was further pursued by the same group from Edinburgh, and resulted in the *Whelk* proof development system [Wiggins *et al*. 91] [Wiggins 92], which is based on a first-order constructive logic with equality, performs proofs in the Gentzen sequent calculus, and extracts logic programs written in Prolog or *Gödel* [Hill and Lloyd 91], a logic programming language that is much closer to the true ideals of logic programming than Prolog.

The system described by [Fribourg 90, 91a] synthesizes deterministic logic programs by coupling program extraction rules to a subset of the inference rules of extended Prolog execution [Kanamori and Fujita 86] [Kanamori and Seki 86]. The synthesized program is de-

fined in a primitive recursive way; moreover, it is guaranteed to be correct wrt the specification, and to terminate. Tail-recursive programs may be synthesized at the expense of the termination guarantee. Little knowledge seems required for the proofs, and the search space seems of manageable size. The automatic generation of simplification lemmas [Fribourg 91b] for syntheses involving proofs-by-induction seems a promising step towards full automation of the mechanism.

Another approach is proposed by [Takayama 87], namely doing the constructive proofs in the typed logical system QJ, and then extracting programs in some intermediate code that is then compiled into Prolog.

2.4.3 Schema-Guided Synthesis of Prolog Programs

As of now, we are not aware of any schema-guided synthesis system for logic programs. We thus quickly enumerate here a few approaches to using schemas for assisting the manual construction of logic programs.

In the field of *logic programming tutors* (for beginners), [Gegg-Harrison 89, 93] proposes a hierarchy of fourteen logic program schemas. These are set in a second-order logic framework. They reflect a divide-and-conquer design strategy, but are already partly instantiated for certain classes of programs.

In the area of *manual or computer-aided program construction* (for experts), [Deville and Burnay 89] and [Deville 87, 90] suggest a divide-and-conquer schema. They also discuss interesting transformation schemas, based on (structural or computational) generalizations.

A similar study is made by [O'Keefe 90], who uses algebraic specifications. The predicates of such specifications can be directly plugged into given logic program schemas. Several schemas may be applicable according to the properties (associativity, commutativity, existence of left identities, and so on) of the identified predicates.

Alternatively, [Lakhotia 89] experiments with what he calls "incorporating programming techniques into Prolog programs". Similarly, [Barker-Plummer 90] discusses a system based on clichés that assists experienced programmers in the construction of Prolog programs. These approaches could be seen as transformation rather than design approaches.

2.5 Conclusions on Program Synthesis from Axiomatizations

The use of deductive inference in program synthesis from axiomatic specifications has given rise to two major approaches: transformational synthesis and proofs-as-programs synthesis. As suggested several times above, these approaches are probably only two facets of the same process. This claim can be substantiated by the ability of several researchers to classify their systems in both sub-categories [Manna and Waldinger 79, 80] [Smith 85] [Kraan *et al.* 92]. Moreover, we have felt the need to create a third sub-category, namely schema-guided synthesis, which is probably yet another facet.

The major problem of deductive approaches to program synthesis is the enormous size of the search spaces, and the lack of power of current theorem provers. For the proving aspects, the use of rippling [Bundy *et al.* 93] and automatic lemma generation [Fribourg 91b] has been suggested for speeding up and further automating proofs-by-induction. For the search space aspects, macroscopic transformation rules, proof-planning via tactics [Bundy 88], and program schemas [Smith 81, 85, 88] have been proposed.

Another problem is, as already said in Chapter 1, that too much emphasis is often put on the synthesis of actual programs in some programming language, which entails taking into account the execution mechanism (and its known deficiencies) and other extra-algorithmic issues. Sometimes, too much emphasis is even put on the synthesis of efficient programs. A

clear separation of correct-algorithm synthesis, algorithm transformation, algorithm implementation (into programs), and program transformation (optimization) helps in overcoming the ensuing problems. Fortunately, more and more researchers focus on algorithm synthesis, leaving transformation and implementations issues to separate research. For instance, the LOPS system of [Bibel 80], the systems of [Smith 81, 85, 88], and the CIP system of [Bauer *et al*. 89] clearly mention algorithms. The notion of logic algorithm (an algorithm expressed in logic) was introduced by [Deville 87, 90] in the context of a methodology for logic program development. This idea has also been advocated and used later for the *Oyster/Clam* synthesis system of [Bundy *et al*. 90] and the *Whelk* system of [Wiggins *et al*. 91] [Wiggins 92].

3 Inductive Inference in Automatic Programming

Inductive inference[1] is not so common in automatic programming. We first define, in Section 3.1, the underlying specification formalism, namely examples. In Section 3.2, we present the major results about inductive inference. Then, in Section 3.3 and Section 3.4, we survey the synthesis of functional programs from examples, and the synthesis of logic programs from examples, respectively. Finally, in Section 3.5, we draw some conclusions on the use of inductive inference in automatic programming.

3.1 Specifications by Examples

We first define a possible language for specifications by examples. Later in this section, we list alternative approaches.

Definition 3-1: A *specification by examples* of a procedure for predicate *r*/*n* consists of a finite set $\mathcal{E}(r)$ of ground examples of *r*/*n*, split into:

- a set $\mathcal{E}^+(r)$ of *positive examples* of *r*/*n* (that is, ground atoms whose *n*-tuples are supposed to belong to $\mathcal{R}$);
- a set $\mathcal{E}^-(r)$ of *negative examples* of *r*/*n* (that is, ground negated atoms whose *n*-tuples are supposed not to belong to $\mathcal{R}$).

Negative examples are also called *counter-examples*. The phrase "an example" stands for "a positive example or a negative example".

We illustrate[2] all this on a few sample specifications.

Example 3-1: The *member(E,L)* relation holds iff term *E* is an element of list *L*. Given a ground value of *L*, this is a non-deterministic relation. A possible specification by positive and negative examples is:

$\mathcal{E}^+$(member) = { member(a,[a]) (E_1)
member(b,[b,c]) (E_2)
member(c,[b,c]) (E_3)
member(d,[d,e,f]) (E_4)
member(e,[d,e,f]) (E_5)
member(f,[d,e,f]) } (E_6)

$\mathcal{E}^-$(member) = { ¬member(a,[]) (C_1)
¬member(d,[b,c]) } (C_2)

Counter-example C_1 suggests that empty lists have no members. C_2 suggests that members must belong to *L*. ♦

In the next two sample specifications, we need two notions. Informally speaking, a *plateau* is a non-empty list of identical elements. A *compact list* is a list of couples, where the first term of a couple, called the *value* of the couple, is different in two consecutive couples, and the second term of a couple, called the *counter* of the couple, is a positive integer.

Example 3-2: The *firstPlateau(L,P,S)* relation holds iff *P* is the maximal plateau at the beginning of the non-empty list *L*, and list *S* is the corresponding suffix of *L*. Given a ground value of *L*, this is a fully deterministic relation. A possible specification by positive and negative examples is:

1. Attention! Inductive inference is not to be confused with deductive proofs-by-induction.
2. Attention! There is a potential ambiguity between "examples of a relation" and "examples that illustrate our purpose". For announcing examples of the latter category, we often use the verb "to illustrate" and the adjective "sample". However, the intended category of examples should always be clear from context

```
E+(firstPlateau)= { firstPlateau([a],[a],[])              (E1)
                    firstPlateau([b,b],[b,b],[])          (E2)
                    firstPlateau([c,d],[c],[d])           (E3)
                    firstPlateau([e,f,g],[e],[f,g])       (E4)
                    firstPlateau([h,i,i],[h],[i,i])       (E5)
                    firstPlateau([j,j,k],[j,j],[k])       (E6)
                    firstPlateau([m,m,m],[m,m,m],[]) }    (E7)

E-(firstPlateau)= { ¬firstPlateau([a],[],[a])             (C1)
                    ¬firstPlateau([b,b],[b],[b])          (C2)
                    ¬firstPlateau([c,d],[c,d],[])         (C3)
                    ¬firstPlateau([e,f,e],[e,e],[f])      (C4)
                    ¬firstPlateau([h,i,i],[i,i],[h]) }    (C5)
```

Counter-example C_1 suggests that plateaus are non-empty. C_2 suggests that maximal plateaus need to be extracted. C_3 suggests that plateaus are lists of identical elements. C_4 suggests that the first plateau of L is not the list of all occurrences of its first element. C_5 suggests that the first plateau of L is not its maximal plateau. ♦

Example 3-3: The *compress(L,C)* relation holds iff C is a compact list of $<v_i,c_i>$ couples, such that the i^{th} plateau of list L has c_i elements equal to v_i. Given a ground value of either parameter, this is a fully deterministic relation. A possible specification by positive and negative examples is:

```
E+(compress)= { compress([],[])                           (E1)
                compress([a],[a,1])                       (E2)
                compress([b,b],[b,2])                     (E3)
                compress([c,d],[c,1,d,1])                 (E4)
                compress([e,e,e],[e,3])                   (E5)
                compress([f,f,g],[f,2,g,1])               (E6)
                compress([h,i,i],[h,1,i,2])               (E7)
                compress([j,k,m],[j,1,k,1,m,1]) }         (E8)

E-(compress)= { ¬compress([b,b],[b,1,b,1])                (C1)
                ¬compress([d,c],[c,1,d,1])                (C2)
                ¬compress([j,k,j],[j,2,k,1]) }            (C3)
```

Counter-example C_1 suggests that plateaus need to be maximally compressed. C_2 suggests that the left-to-right order of appearance of the plateaus in L determines the order of appearance of the couples in C, and not, say, the alphabetical order. C_3 suggests that non-adjacent plateaus of the same value are compressed separately. ♦

Example 3-4: The *sum(L,S)* relation holds iff S is the sum of the elements of integer list L. Given a ground value of L, this is a fully deterministic relation. A possible specification by positive and negative examples is:

```
E+(sum) = { sum([],0)                                     (E1)
            sum([7],7)                                    (E2)
            sum([4,5],9)                                  (E3)
            sum([2,3,1],6) }                              (E4)

E-(sum) = { ¬sum([],32767)                                (C1)
            ¬sum([7],9)                                   (C2)
            ¬sum([4,5],4) }                               (C3)
```

Counter-example C_1 suggests that the sum of the empty list is not different from 0. C_2 suggests that the sum of a singleton list is not different from the first element of that list. C_3 suggests that the sum of a list of at least 2 elements is not the first element of that list. ♦

Example 3-5: The *sort(L,S)* relation holds iff *S* is an ascendingly ordered permutation of integer list *L*. Given a ground value of *L*, this is a fully deterministic relation. A possible specification by positive and negative examples is:

$\mathcal{E}^+$(sort)= { sort([],[]) (E_1)
sort([1],[1]) (E_2)
sort([3,2],[2,3]) (E_3)
sort([6,4,5],[4,5,6]) } (E_4)

$\mathcal{E}^-$(sort)= { ¬sort([3,2],[3,2]) (C_1)
¬sort([6,4,5],[4,6,5]) (C_2)
¬sort([6,4,5],[5,6]) (C_3)
¬sort([6,4,5],[4,5,6,7]) } (C_4)

Counter-example C_1 suggests that a sorted list is not always equal to the unsorted list. C_2 suggests it doesn't always suffice to permute the first two elements. C_3 and C_4 suggest that the elements of a sorted list are exactly the same than those of the unsorted list. ♦

It should be noted that Definition 3-1 is not the only possible definition for specifications by examples. It is too much geared towards the synthesis of algorithms. So let's generalize it to the larger framework of empirical machine learning from examples. Specification approaches vary according to the following criteria:

- *multiplicity of examples*: sometimes, only one example is required, but most often, more than one example is expected;
- *language of examples*:
  - *groundness*: examples are mostly ground literals, but one could also consider any kinds of literals; for instance, *sum*([*X*],*X*) would be a useful example;
  - *number of predicates*: often, a single literal with predicate *r*/*n* constitutes an example; another approach is a conjunction of literals with predicates other than *r*/*n* that describe the attributes of an example (then called an *instance description*) of a relation *r*/*n* (then called a *concept*, where usually *n*=1); for example, the positive instance *tweety* of the concept *bird*/1 could be described as follows:

$$mouth(tweety,beak) \wedge legs(tweety,2) \wedge skin(tweety,feathers) \wedge utterance(tweety,sings) \wedge color(tweety,yellow)$$

  - *directionality*: examples are often given in a relational form, but the functional form (which lists *input*/*output examples*) is also very common; for instance, *compress*([*c*,*d*]) = [*c*,1,*d*,1] is an input/output example leading to a function that can only compress lists, but not decompress compact lists;
- *kinds of examples*: the presence of positive and negative examples is commonly expected, but sometimes, only one of these two kinds of examples is required.

Other criteria depend on the learning mechanism, but do not define specifications:

- *moment of choice of examples*: the two extreme approaches are choosing all the examples before the learning, respectively choosing all the examples on-the-fly during the learning, as the need for more examples arises;
- *formulation of examples*: examples are formulated either directly as examples by the specifier, or indirectly as classification queries by the learning mechanism; in the latter case, the specifier has to classify an example as a positive or negative example.

Finally, some criteria depend on the circumstances under which examples are chosen:

- *specifying agent*: the specifier is either a human being, or some automated device, such as a robot, a sonde, a catheter, and so on;
- *status of intended concept*: at the moment of choosing the examples, the intended concept is either known (even if only informally), or unknown; for instance, the *compress*/2 relation was known for the elaboration of the specification in Example 3-3, but the protein structure of a molecule is unknown during the choice of examples;
- *consistency of examples*: either examples are known or assumed to be consistent with the intended concept, or examples are known or assumed to be possibly inconsistent with the intended concept.

All these criteria allow a precise classification of settings for learning from examples.

Anyway, whatever the actual setting, the advantages of specifications by examples are the following:

+ *naturalness*: human beings often resort to examples in order to explain some concept; moreover, examples are easy to elaborate and easy to understand;
+ *conciseness*: examples are a concise[3] way of intensionally specifying some concept, because the actual computations are abstracted by a black-box model.

The major disadvantage of specifications by examples is:

– *incompleteness* and *ambiguity*: a finite set of examples cannot completely and unambiguously specify an infinite relation; in practice, one even expects the number of given examples to be orders of magnitude less than the number of all possible examples; for instance, the total ordering underlying the *sort*/2 relation is mentioned nowhere; the same holds for the summation operator underlying the *sum*/2 relation; the kind of determinism of a relation is notoriously hard to convey by examples; in Example 3-2, the sub-concept of plateau is not entirely clear from the examples, as nothing explicitly says that plateaus have to be maximal, or that it is only the difference between two successive elements of L that determines where the break into P and S occurs, and not some other strange criterion that is based, say, on the notions of vowels, consonants, or positions within the alphabet.

From this drawback, it is obvious that, given a specific learning mechanism and a human specifier, there is a definite need for a methodology for choosing "good" examples.

3.2 Inductive Inference

Inductive inference is the process of formulating a general rule from incomplete information, such as examples. This ability is a fundamental component of human intelligence. *Learning* is the process of increasing one's knowledge or skill. This is a broader problem than inductive inference, though techniques of the latter are often used in learning. The branch of artificial intelligence research that is about learning is called *machine learning*. In this section, we only concentrate on the sub-branch of machine learning that focuses on the so-called *empirical learning from examples*, which relies heavily on inductive inference.

In Section 3.2.1, we suggest a terminology for the components of an empirical learning system. Then, in Section 3.2.2, we define rules of inductive inference, and survey the usage of these rules in empirical learning from examples in Section 3.2.3. Finally, in Section 3.2.4, we define the niche of algorithm synthesis from examples within empirical learning from examples, and give pointers to the literature in Section 3.2.5.

3. This needs to be relativized, though. First, large-size data-structures go against conciseness. For instance, specifying 3D-matrix-multiplication by examples is not really concise, not to mention the tediousness of doing so (assuming a human specifier). Second, large numbers of examples also go against conciseness. Expansion factors are very small in such cases, but one should not forget that such large and/or numerous examples are sometimes all that is available for learning.

3.2.1 Components of a Learning System

We first propose a terminology for the components of an empirical learning system.

A *teacher* (*trainer, instructor*) chooses *instance descriptions* (*examples*) of instances of the *intended concept*, and *presents* them to the *learner*. There are basically two ways of presenting examples: *all-at-once*, and *chunk-wise* (for instance, *one-by-one*).

Sometimes, the learner asks an *oracle* (*informant*) for classification of examples it invents itself. The oracle is usually the teacher, but is sometimes automated via reliance on other information, such as *background knowledge*.

The learner periodically produces a *hypothesis* (*conjecture, guess, concept description*).

This terminology is sufficient for our purpose. But note that a complete and general learning system would have many more components, such as a *blackboard*, an *example selector*, a *critic*, and so on. An ideal architecture is described by [van Lamsweerde 91].

3.2.2 Rules of Inductive Inference

Logical formulations of inductive inference rules were developed by [Genesereth and Nilsson 88] and by [van Lamsweerde 91]. This chapter is loosely based on these papers.

Definition 3-2: Given background knowledge $\mathcal{B}$ and a set of examples $\mathcal{E}$, a hypothesis H is an *inductive conclusion* of $\mathcal{B}$ and $\mathcal{E}$ iff the following four conditions hold:

(1) $\mathcal{B} \cup \{H\} \models \mathcal{E}$ (the hypothesis "explains" the examples);

(2) $\mathcal{B} \cup \mathcal{E} \not\models \neg H$ (the hypothesis is consistent with the given data);

(3) $\mathcal{B} \not\models H$ (the hypothesis is not a consequence of the background knowledge);

(4) $\mathcal{B} \not\models \mathcal{E}$ (the examples are not redundant with the background knowledge).

Note that inductive inference is not necessarily sound: $\mathcal{B} \cup \mathcal{E} \models H$ does not hold in general. But, on the other hand, not every induction is unsound.

A list of atomic inductive inference rules includes *standard generalization*, *deleting-a-conjunct*, *adding-a-disjunct*, *replacing-a-constant-by-a-variable*, *splitting-a-variable-into-several-variables*, *weakening-an-implicant*, *extending-the-domain-of-a-variable*, *closing-an-interval*, and *climbing-a-specialization-tree*. More elaborate inductive inference rules for complex formulas are based on the fundamental notions of *generalization* and *msg* (*most-specific-generalization*): these are surveyed in detail in Chapter 7 and Chapter 10.

Also, since there are many possible inductive conclusions for any background knowledge and example set, it is necessary to prune the generalization search space so as to get "useful" inductive conclusions. Interesting approaches are based on *induction biases*:

- *semantic bias* restricts the vocabulary available for hypotheses; for instance, the *model maximization technique* prefers hypotheses that have a maximal number of models, and *conceptual bias* restricts the vocabulary to a predefined set of predicates, called the *basis set*; in this context, a *selective rule* of inductive inference only refers to the predicates appearing in the given examples, whereas a *constructive rule* yields generalizations that "invent" predicates not appearing in the given examples;
- *syntactic bias* restricts the language of hypotheses; for instance, (existential) conjunctive hypotheses and disjunctive normal form hypotheses represent very common search spaces, both in propositional calculus and in first-order logic.

Other approaches are based on preference criteria for utility or efficiency measures.

We here only focus on *instance-to-class* generalization. Other kinds of generalization are *class-to-class* generalization and *part-to-whole* generalization.

3.2.3 Empirical Learning from Examples

The aim of empirical learning (concept formation) from examples is the elaboration of a concept description from multiple positive and negative examples, when little—or no—background knowledge is available. The inferred hypothesis must be satisfied by all the positive examples, but by none of the negative examples. In the sequel, "learning" stands for "empirical learning". In this section, we discuss concept classes, examples, presentations, hypotheses, possible identification criteria, a classification of learning mechanisms, and actual learning mechanisms. We actually simultaneously summarize both the more theoretical concerns (such as learnability) of the inductive inference community, and the more practical concerns (such as actual mechanisms) of the machine learning community. Hence, "learning" also stands for "identification".

Concept Classes, Examples, and Presentations

There are many classes of concepts: functions, relations, sequences, (stochastic) languages, and so on. The kinds of examples used for illustrating these concept classes, as well as the nature of admissible presentations of these examples, vary accordingly. Moreover, the kinds of hypotheses for describing elements of these concept classes also vary accordingly.

Example 3-6: The concept class aimed at by Definition 3-1 is the set of relations over any domain. Examples of a relation are elements of its graph, and any (possibly infinite) sequence of such examples constitutes an admissible presentation iff this sequence lists all and only the elements of the graph of the relation. Hypotheses could be expressed as algorithms, programs, Turing machines, automata, and so on.

Example 3-7: In *grammatical inference*, the concept class aimed at is the set of languages over some alphabet. Examples of a language are sentences of that language. Hypotheses could be expressed as grammars, acceptors, regular expressions, and so on.

Restrictions on admissible presentations may have to be imposed, such as computability (below some complexity threshold), or ordering according to some total order. A *positive presentation* lists all and only the positive examples, whereas a *complete presentation* lists all the positive and negative examples. A *presentation by informant* queries an oracle for classification of examples as positive or negative examples. A *mixed presentation* combines complete presentation and presentation by informant. Repetition is usually allowed in admissible presentations.

Hypotheses

Learning can be abstracted as the search through a state space, where states correspond to hypotheses, and operators correspond to the application of rules of inductive inference.

Various constraints may be imposed on the hypothesis space.

Example 3-8: The hypothesis space is often supposed to be constrained by a conceptual bias and a syntactic bias: an *acceptable hypothesis* satisfies both biases. A hypothesis is *characteristic* for a set $\mathcal{E}$ of examples iff it is satisfied by all the positive examples of $\mathcal{E}$. A hypothesis is *discriminant* for $\mathcal{E}$ iff its negation is satisfied by all the negative examples of $\mathcal{E}$. An acceptable hypothesis is *admissible* (*consistent*) for $\mathcal{E}$ iff it is characteristic for $\mathcal{E}$ and discriminant for $\mathcal{E}$. The *version space* for a set of examples $\mathcal{E}$ is the set of all admissible hypotheses (versions) for $\mathcal{E}$.

Moreover, *goodness orderings* may be defined for comparing hypotheses. Such orderings usually are partial orders, and are often required to be computable. A goodness ordering is *sample-independent* iff the order between two hypotheses is preserved for any example set.

Example 3-9: For instance, a version space is usually organized according to a partial order under which hypothesis H_1 is *less general* than H_2 iff H_1 logically implies H_2. This is a sample-independent ordering, but decidable only in specific settings. The set of *minimal* (*maximally specific*) hypotheses (no hypothesis in $\mathcal{V}$ is less general than these) of $\mathcal{V}$ is called the *lower boundary set* of $\mathcal{V}$, and the set of *maximal* hypotheses (every hypothesis in $\mathcal{V}$ is less general than these) of $\mathcal{V}$ is called the *upper boundary set* of $\mathcal{V}$. A version space $\mathcal{V}$ is *well-formed* iff every ascending chain of maximal length in $\mathcal{V}$ starts from a hypothesis that is minimal in $\mathcal{V}$ and ends in a hypothesis that is maximal in $\mathcal{V}$.

Other goodness orderings are based on some measure of the *size* of hypotheses, or on some measure of the *simplicity* of hypotheses, or on probabilistic measures, and so on.

The size of the hypothesis space is typically exponential, even when constrained and partially ordered. However, the size of a search space never prejudices the existence of efficient learning mechanisms for that space. Of course, the most efficient mechanism is usually very specific to its search space, and hence difficult to adapt into a more general mechanism.

Example 3-10: Version spaces are of a size exponential in the size of the basis set, not to mention the syntactic bias. Fortunately, well-formed version spaces (for instance, when the basis set is finite) have the interesting property that they are indeed bound by their boundary sets, which provides for an admissibility check that is independent of the examples. See below for an application of this property.

Identification Criteria

An identification criterion is a criterion that defines under what circumstances learning is successful in identifying the intended concept (that is, in producing a hypothesis that is correct wrt the intended concept). There are two main identification criteria.

First, the seminal work by [Gold 67] has introduced the criterion of *identification-in-the-limit*. Learning is viewed as an infinite process of example presentations and hypothesis productions whose limiting behavior is used as a criterion of success.

Definition 3-3: A learner L correctly *identifies* an intended concept C *in the limit* iff the infinite sequence of hypotheses produced by L, say $H_1, H_2, \ldots$, is such that there exists some number n such that $H_n = H_m$ for every $m>n$, and such that H_n is "correct" wrt C.

Note that there is no way that L can tell that it actually has identified C, because the next example might or not provoke a revision of the last hypothesis. An important limiting result is that the class of total recursive functions is not identifiable-in-the-limit: this means that there can't be a universal mechanism of program synthesis from incomplete information. However, the class of primitive recursive functions is identifiable-in-the-limit. Moreover, given positive examples only, the class of finite languages is identifiable-in-the-limit, but no class containing all finite languages and at least one infinite language is identifiable-in-the-limit [Gold 67].

A possible way of achieving identification-in-the-limit is the *identification-by-enumeration* learning mechanism: after each presentation of a new example, one systematically traverses the entire search space until a hypothesis is found that is satisfied by all the examples presented so far. This requires computability of the enumeration and of the satisfiability of hypotheses wrt examples. This is however an extremely impractical mechanism because of the (typically exponential) size of the search space. Improvements usually consist in defining a partial order on the search space, which then allows pruning of uninteresting branches, and sometimes even suggesting plausible replacements for rejected hypotheses. Other improvements are based on heuristic measures of the goodness of hypotheses.

Years later, Leslie G. Valiant has introduced *PAC (probably-approximately-correct) identification* [Valiant 84, 85] [Pitt and Valiant 88], which is today's most-used learning model. The idea is to weaken the tight constraints of identification-in-the-limit by allowing the final hypothesis to be "nearly correct" wrt the intended concept. Moreover, the actual learning process is usually based on a finite presentation, whereas admissible presentations are mostly infinite. Other motivations are the need for polynomial-time learning, and the need for tolerance to erroneous examples: hence a probabilistic specification approach.

Definition 3-4: A learner *PAC-identifies* an intended concept iff an arbitrary hypothesis that is consistent with the examples presented so far can be guaranteed with high probability to be consistent with most of the examples that are to be presented in the sequel.

The main result is that whatever the probability distributions from which the examples are drawn, a hypothesis distinguishing the positive examples from the negative examples with controllable error can be inferred easily, except if the hypothesis language is too expressive.

Many other identification criteria have been defined. An *identification type* is the class of all the sets of hypotheses that are identifiable according to some given identification criterion. A wealth of learnability results order the identification types by set-inclusion into an interesting partially-ordered set. Data presentation has a major impact on these results. Positive presentation seems to be afflicted with negative results for identification-in-the-limit.

Finally, note that identification is not the only inductive inference problem. Another problem is *prediction* (*extrapolation*), where the n^{th} value of a sequence is to be predicted from the n–1 first values of that sequence. Criteria of successful prediction have been defined, but we do not discuss this topic here.

Classification of Learning Mechanisms

We here focus on stepwise learning, whose purpose is to have monitoring points between steps so that the learning process can be assessed in terms of correctness and progression.

Definition 3-5: *Stepwise learning* is performed by a sequence of steps, each producing a hypothesis that is designed from the previous hypothesis and from (some of) the examples presented so far.

Note that this definition does not preclude the sequence of steps to be (partly) defined by a loop over the same step. This definition also does not mention how and when examples are presented. Stepwise learning can be divided into two complementary sub-categories:

Definition 3-6: *Incremental learning* is stepwise learning where each hypothesis may be the final one. *Non-incremental learning* is stepwise learning where only the last hypothesis is supposed to be the final one, the other hypotheses being temporary.

Incremental learning is often used in conjunction with a one-by-one presentation of the examples (prior to each step, a single example is presented), and a single step that is iterated over after each presentation. In the sequel, we equate incremental learning with this particular case. Non-incremental learning is often used in conjunction with an all-at-once presentation of the examples (prior to the first step, all the examples are presented), and multiple steps that are each executed only once. In the sequel, we equate non-incremental learning with this particular case. These two approaches have complementary pros and cons:

- considering that an example embodies several kinds of information, incremental learning has to extract all these kinds of information from the examples presented so far, whereas non-incremental learning may extract only one kind of information from every example at each step;

- incremental learning is prone to be sensitive to the order of presentation of the examples, in the sense that different final hypotheses may emerge from different orders of presentation, whereas non-incremental learning is usually insensitive to the order of presentation of the examples, thus turning the latter into a fully declarative specification; non-incremental learning usually first performs some example ordering;
- incremental learning only consumes as many examples as are needed, because it relies on an example stream, whereas non-incremental learning requires all the examples to be ready prior to learning, which may result in a useless computational overhead, or in teacher inflexibility.

Choosing one of these strategies is obviously a very delicate design decision.

Other useful characterizations of learning mechanisms are as follows:

Definition 3-7: *Algorithmic learning* is performed by the sole execution of a learning algorithm. *Heuristic learning* is at least partially based on heuristics.

Definition 3-8: *Interactive learning* involves experiment generation and queries to an oracle. *Passive learning* is non-interactive learning.

We now establish a list of useful classification criteria for stepwise learning mechanisms. They amount to restricting identification types, but the reward is a tight knowledge about, and control over, the evolution of the learning process:

- *iterative learning* is incremental learning where every new hypothesis is designed solely from the most recently presented example and from the previous hypothesis;
- *monotonic learning* is stepwise learning where every new hypothesis covers no less conclusions than the previous hypothesis;
- *consistent learning* is stepwise learning where every new hypothesis is consistent with all examples presented so far;
- *conservative learning* is incremental learning where a hypothesis is different from its predecessor only if the latter is inconsistent with the last presented example;
- *reliable learning* is stepwise learning that converges to the correct hypothesis, whenever it does converge;
- *finite learning* is stepwise learning that stops (has converged) iff two successive hypotheses are the same (this removes the uncertainty about the convergence of identification-in-the-limit);

Combinations of these criteria are of course possible.

Some concepts are learnable only via non-monotonic learning [Jantke 91]. Other concepts are learnable only via inconsistent learning [Lange and Wiehagen 91]. A host of learnability results inserts the corresponding refined identification types into the partially-ordered set of identification types [Jantke 88, 91].

Learning mechanisms may be compared in terms of various criteria. Possible comparison criteria are as follows:

- given an identification criterion I and an example presentation method P, a learning mechanism is *more powerful* than another one wrt I and P iff its class of hypotheses that can be correctly identified wrt I and P properly contains the corresponding class of the other mechanism;
- a learning mechanism is *(strictly) more data-efficient* than another one iff it converges (strictly) "faster" than the other one, whenever that one converges; a learning mechanism is *optimally data-efficient* iff no mechanism is strictly more data-efficient than it;
- the overall *number of distinct hypotheses* and the *number of mind changes* (whenever two successive hypotheses are distinct) are other measures for comparing learning mechanisms;

and so on.

Example 3-11: Identification-by-enumeration is an incremental, consistent, conservative, and optimally data-efficient learning mechanism [Gold 67].

Learning Mechanisms

Empirical learning research has basically taken two approaches [van Lamsweerde 91]:

- the *data-driven approach* starts from examples only; the examples are assumed consistent wrt the intended concept;
- the *approximation-driven approach* (also called the *model-driven approach*) starts from examples and a set of approximate hypotheses about the intended concept; approximate hypotheses are incorrect, or incomplete, or both;

Let's briefly discuss these approaches.

The data-driven approach features incremental learning, with (by definition) an iteration over the set $\mathcal{E}$ of given examples, where each presentation of an example leads to an update of the current hypothesis (set) wrt that example. There are three main strategies for reducing the version space to a hypothesis (set) that is admissible for $\mathcal{E}$:

- *depth-first search*: at each presentation of an example, the search for an admissible hypothesis considers only one hypothesis, which is adapted according to the presented example. The drawbacks are the need for managing backtracking, and the need for re-checking the admissibility of the new hypothesis wrt all examples presented so far that are of a sign opposite to the one of the last-presented example;
- *breadth-first search*: at each presentation of an example, the search for an admissible hypothesis considers several hypotheses in parallel, which are adapted according to the presented example. The considered hypotheses are those of the lower boundary set. The successive hypotheses are monotonically increasing in generality, so there is no need for re-checking whether the new hypothesis is characteristic wrt all positive examples presented so far. The drawback is the need for re-checking whether the new hypothesis is discriminant wrt all negative examples presented so far. A variant of this strategy, for well-formed version spaces only, achieves symmetry between the handling of positive and negative examples by simultaneously considering the lower and upper boundary sets at each presentation: this leads to the *candidate-elimination mechanism* of [Mitchell 81]. Convergence of the ascending and descending hypothesis series to the same unique limit is guaranteed, and admissibility re-checking is useless;
- *heuristic search*: in order to cope with extremely large version spaces, heuristic preference functions may be introduced; Michalski's *star algorithm* is a sample application of this idea, but we do not develop this here.

These three strategies differ by the representative subset of the version space they build. Ultimately, the minimal hypotheses of the inferred hypothesis (set) are retained, because they are satisfied by the potentially smallest possible set of examples unrelated to the intended concept. The data-driven approach has a low tolerance to erroneous examples.

The approximation-driven approach features non-incremental learning, with an iteration over the given approximations. There are two strategies for doing so:

- *bottom-up search*: if the given approximations are not general enough, then use any of the data-driven learning algorithms for generalizing an approximation into a set of maximally specific hypotheses that are admissible wrt all the given examples;
- *top-down search*: if the given approximations are too general, then use the dual of any of the data-driven learning algorithms for specializing an approximation into a set of maximally specific hypotheses that are admissible wrt all the given examples.

The inferred hypotheses are collected into a hopefully representative subset of the version space. The approximation-driven approach has some tolerance to erroneous examples. The non-incrementality is often seen as a disadvantage of the approximation-driven approach.

Empirical learning from examples usually requires a sufficient number of examples for successful learning of the intended concept. Moreover, the shown approaches are very sensitive to the quality and ordering of examples. Investigated optimization techniques include the localization and pruning of forbidden features, hill-climbing for searching locally optimal hypotheses, jumping-to-conclusions based on plausible features, and so on.

3.2.4 Algorithm Synthesis from Examples as a Niche of Machine Learning

The synthesis of algorithms from examples is a machine learning task as it falls into the category of empirical learning from examples. Indeed, let's state the objectives of both fields:

- the aim of empirical learning from examples is the elaboration of a concept description from multiple positive and negative examples of the intended concept;
- the aim of synthesis from examples is the design of an algorithm from multiple positive and negative examples of its intended functionality.

The renaming substitutions to pass from one objective to the other one are obvious.

But algorithm synthesis from examples is a highly specialized niche within empirical learning from examples. Table 3-1 summarizes the differences between the concerns of algorithm synthesis (as we view it) and the mainstream concerns (so far) of empirical learning.

Table 3-1: Our view of algorithm synthesis from examples as a niche of empirical learning

	Empirical learning	Algorithm synthesis
Intentions	any	relations
Class of hypotheses	any	recursive algorithms
Specifying agent	any agent	often a human being
Status of intended concept	sometimes unknown	always known
Consistency of examples	any attitude	assumed consistent
Number of predicates in examples	≥ 1	1
Rules of inductive inference	mostly selective	necessarily constructive
Correctness of hypotheses	any attitude	always crucial
Existence of hypothesis schemas	hardly	yes, many
Number of correct hypotheses	usually only a few	always many

Indeed, for algorithm synthesis, we are here only interested in recursive algorithms for relations, whereas most concept descriptions are non-recursive. Moreover, we are here mostly interested in human specifiers who know—even if only informally—the intended relation, and who are assumed to choose only examples that are consistent with the intended relation. But this setting constitutes only a particular case of empirical learning. Then, as in Definition 3-1, examples of an intended relation only involve one predicate, whereas instance descriptions usually involve several predicates. Algorithm synthesis from examples thus necessarily has to focus on constructive rules of inductive inference, whereas empirical learning can explore vast research areas by just using selective rules. Also, a synthesized algorithm is

only useful if it is totally correct wrt its intended relation, whereas a learned concept description may be useful despite some deficiencies. Finally, algorithms are highly structured, complex entities that are designed according to some strategy (such as divide-and-conquer, generate-and-test, …), whereas concept descriptions can have virtually any format (unless constrained by some induction bias): algorithm synthesis can thus be effectively guided by a schema (see Chapter 8) reflecting some design strategy. The existence of many such schemas, and the existence of many choices within the corresponding design strategies, entail the existence of many correct algorithms (even if they have vastly different complexities) for a given intended relation. For instance, the sorting problem may be solved by algorithms such as Insertion-Sort, Merge-Sort, Quick-Sort, and so on. Such a multiplicity of correct solutions is unusual elsewhere.

In other words, algorithms are a highly specialized kind of concept descriptions. We believe that algorithm synthesis thus deserves a highly specialized set of mechanisms, rather than off-the-shelf ones from empirical learning. For algorithm synthesis, we thus adopt the following terminology: we say "specifier" instead of "teacher", "synthesizer" instead of "learner", "example" instead of "instance description", "relation" instead of "concept", "algorithm" instead of "concept description", and "synthesis" instead of "learning".

3.2.5 Pointers to the Literature

An excellent historical overview of the role of inductive inference in scientific discovery is given by [Muggleton 91]. Comprehensive introductions to, and surveys of, inductive inference research are those of [Angluin and Smith 83], [Angluin 84], [Biermann 86], and [Jantke 89]. In this theoretical field, there are highly specialized workshops such as AII (*Analogical and Inductive Inference*), ALT (*Algorithmic Learning Theory*), and COLT (*COmputational Learning Theory*). Publications for inductive inference results are *New Generation Computing* and *Machine Learning*.

A comprehensive survey of machine learning techniques is proposed by [van Lamsweerde 91]. In this more practical field, there are conferences such as ICML (*International Conference on Machine Learning*), ECML (*European Conference on Machine Learning*, formerly EWSL: *European Working Session on Learning*), and the various artificial intelligence conferences. Publications for machine learning results are basically the same as above.

Unfortunately, there seems to be little cross-fertilization between these two communities: "abstract results proliferate uselessly, while the concrete results produce little or nothing of significance beyond their very narrow domains" [Angluin and Smith 83]. Moreover, terminologies often diverge. Closer collaboration and unification of the frameworks seem thus highly desirable. Especially the incorporation of domain knowledge into the theoretical results of inductive inference seems a must, because the exponential complexities of the learning mechanisms suggested by these results make them unusable in practice, and because such knowledge is after all available to human learners.

Applications of inductive inference and machine learning are numerous. A most promising application of the technology is the accelerated bootstrapping of expert systems by automated acquisition of rules from example sets, rather than by interviewing human experts. Also, advanced user-interfaces allow non-expert computer users to specify by examples what they want the computer to do. For instance, the *Query-by-Example* system of [Zloof 77] allows easy database querying. Or the *EP* systems of [Waterman *et al*. 84] assist at the operating system level. Finally, as shown in Section 3.2.4, algorithm (or program) synthesis is an application, too. The next two sections survey the synthesis of functional programs from examples, and the synthesis of logic programs from examples, respectively.

3.3 Functional Program Synthesis from Examples

The synthesis of functional programs (LISP programs, say) was an area of intense research during the 1970s. Since the inferred concept description is a functional program, it suffices to present the examples in the functional form: examples are then called *input/output-examples*, and abbreviated *I/O-examples*, or simply examples for the purpose of this section. Recall the important limiting result of [Gold 67], which says that the class of total recursive functions is not identifiable-in-the-limit: this means that there can't be any universal mechanism of program synthesis from incomplete information. However, [Gold 67] also shows that a function can be inferred from examples if it belongs to a class of enumerable functions with a decidable halting problem. A detailed and insightful survey was made by [Smith 84]. The general synthesis surveys of [Biermann *et al*. 84b] [Biermann 92] also include sections on synthesis from examples. This present survey is organized as follows. First, Section 3.3.1 surveys synthesis from traces, which appears later to be a useful component of synthesis from examples. Then, we survey algorithmic and heuristic approaches to synthesis from examples in Section 3.3.2 and Section 3.3.3, respectively.

3.3.1 Synthesis from Traces

A *trace* is a sequence of instructions executed by a program on some given input data. Traces are often used by human beings to explain an existing algorithm. For instance, the protocol of using a telephone is easily described by sample traces.

Biermann has developed a general algorithmic mechanism of learning-from-traces (the *Trainable Turing Machine*) [Biermann 72], and has applied it to program synthesis, such as the trainable desk calculator of [Biermann and Krishnaswamy 76], which identifies the intended program in the limit. The given traces are equal to the traces the synthesized program produces on the same input data. A sample synthesis appears in Example 3-12 below. Other research on synthesis from traces includes the work of [Bauer 79] on extending Biermann and Krishnaswamy's results.

Traces are however a very tedious and error-prone specification formalism. Worse, traces oblige the specifier to already know the desired algorithm, which often goes counter the setting of algorithm synthesis. All this somewhat discredits synthesis-from-traces as an approach to automatic programming.

3.3.2 Algorithmic Synthesis from Examples

Algorithmic synthesis from examples is devoid of heuristics. All the surveyed mechanisms proceed in two steps:

(1) *trace generation*: trace(s) are generated from the input/output example(s);

(2) *synthesis from traces*: the trace(s) are generalized into a recursive program.

This approach gives new value to synthesis-from-traces. This idea of decomposition seems to stem from [Siklóssy and Sykes 75]. A problem solver, called LAWALY, generates traces from multiple examples, in a heuristic way. The synthesizer proper, called SYN, fully automatically generalizes the traces into LISP-like recursive programs. There is however no concern about correctness, nor about the targeted class of programs.

We here discuss the two best-developed approaches to this idea, namely Biermann's function merging mechanism, and Summers' recurrence relation detection mechanism. Both reflect passive, non-incremental, two-step, consistent synthesis from positive, ground, presynthesis I/O-example(s) that are selected by an agent who knows the intended function.

Biermann's Function Merging Mechanism

Biermann has also applied this approach, but he used his own synthesis-from-traces mechanism [Biermann 72] for the second step. The resulting synthesis mechanism for so-called regular LISP programs (with only one parameter, and where the only predicate, if any, is *atom*) is described by [Biermann 78, 84a]. Let's first illustrate it on a simple synthesis (taken from [Biermann 92]).

Example 3-12: Suppose we want a LISP function for reversing S-expressions. The following example specifies this:

```
reverse(((a•b)•c)) = (c•(b•a))
```

The synthesis goes as follows.

First, the output Y is uniquely decomposed—in an algorithmic way—by applying the basic functors *car*, *cdr*, and *cons* on the input X:

```
Y = cons(cdr(X),cons(cdr(car(X)),car(car(X))))
```

This decomposition can be re-expressed as the following trace:

```
Y = f1(X)
f1(X) = cons(f2(X),f3(X))
f2(X) = f4(cdr(X))
f3(X) = f5(car(X))
f4(X) = X
f5(X) = cons(f6(X),f7(X))
f6(X) = f8(cdr(X))
f7(X) = f9(car(X))
f8(X) = X
f9(X) = X
```

Second, the synthesis-from-traces mechanism proceeds by merging the obtained functions into a minimal number of functions that preserve the original computations. If a merged function is multiply defined, then a predicate generator searches for discriminants. This works here as follows. The functions f_1,f_4,f_5,f_8, and f_9 may be merged into a unique function *reverse*. The predicate generator infers that the body of f_4, f_8, and f_9 is applicable iff the parameter is an atom. Hence the following definition of *reverse*:

```
reverse(X) = cond( (atom(X) X) (T cons(f2(X),f3(X))) )
```

Similarly, function f_6 may be merged (unconditionally) into f_2, and function f_7 may be merged (unconditionally) into f_3:

```
f2(X) = reverse(cdr(X))
f3(X) = reverse(car(X))
```

The functions *reverse*, f_2, and f_3 constitute the synthesized program. ♦

This function merging mechanism usually works from any single "significant" positive example. It is however very costly in computations (namely of a complexity that is exponential on the size of the target program), because of its identification-by-enumeration approach. It finitely identifies the intended program in the limit. It thus performs consistent, conservative, and optimally data-efficient synthesis. Moreover, it can synthesize any program in the class of regular LISP programs (a strict sub-set of LISP programs).

As [van Lamsweerde 91] observes, this kind of synthesis from examples actually is a precursor to EBL (*Explanation-Based Learning*), another branch of machine learning. Indeed,

the goal of EBL in general [DeJong and Mooney 86], and EBG (*Explanation-Based Generalization*) in particular [Mitchell *et al*. 86], is the elaboration of a concept description from a very small number of examples (typically a single one), in the presence of much background knowledge. EBL proceeds by first seeking an explanation of why the example describes an instance of the intended concept (using the background knowledge), and then generalizing that explanation. Rules of deductive, abductive, and analogical inference are typically used in EBL, in contrast to the rules of inductive inference used in empirical learning: EBL reflects thus analytic learning.

Summers' Recurrence Detection Mechanism

The THESYS system of [Summers 77] reflects another solution to the two-step approach that is based on traces. Let's first illustrate it on a simple synthesis.

Example 3-13: Suppose we want a LISP function for reversing S-expressions. This may be specified as follows:

```
reverse(a) = a                                  (E1)
reverse((b•c)) = (c•b)                          (E2)
reverse(((d•e)•f)) = (f•(e•d))                  (E3)
reverse((((g•h)•i)•j) = (j•(i•(h•g)))           (E4)
```

The synthesis goes as follows.

First, each sample output is uniquely decomposed—in an algorithmic way—by applying the basic functors *car*, *cdr*, and *cons* on the corresponding input: [4]

$f_1(X) = X$

$f_2(X) = cons(cdr(X),car(X))$

$f_3(X) = cons(cdr(X),cons(cdar(X),ca^2r(X)))$

$f_4(X) = cons(cdr(X),cons(cdar(X),cons(cda^2r(X),ca^3r(X))))$

where function f_i corresponds to example E_i. Similarly, predicates are generated to recognize terms of the same structure than each sample input:

$p_1(X) = atom(X)$

$p_2(X) = atom(car(X))$

$p_3(X) = atom(ca^2r(X))$

$p_4(X) = atom(ca^3r(X))$

where predicate p_i recognizes terms of the same structure than the input term of example E_i.

Second, recurrence relations are sought among the decompositions:

$f_1(X) = X$

$f_i(X) = cons(cdr(X),f_{i-1}(car(X)))$ (i>1)

Similarly, recurrence relations are sought among the structure recognition predicates:

$p_1(X) = atom(X)$

$p_i(X) = p_{i-1}(car(X))$ (i>1)

These recurrence relations are then plugged by the so-called *Basic Synthesis Theorem* into a LISP program schema that reflects a divide-and-conquer design strategy. This yields the following LISP program:

4. The function cd^nr denotes the composition of n applications of the function *cdr*. The function cd^na^mr denotes the composition of n applications of the function *cdr* composed to the composition of m applications of the function *car*.

```
reverse(X) = cond( (atom(X) X)
                   (T cons(cdr(X),reverse(car(X)))) )
```

This completes the synthesis. Note that this program is not doubly recursive like the one synthesized in Example 3-12: the reasons are (*i*) that the examples given here incorrectly suggest that the input S-expressions are always left-linear trees, and (*ii*) that Biermann's mechanism looks for the smallest possible merger of the decomposition trace. Indeed, Biermann's mechanism would find the above program by merging f_4 and f_8 not into *reverse*, but into each other, which however doesn't yield the smallest merger. ♦

This recurrence detection mechanism requires multiple positive examples, which overcomes the potential ambiguity problems of single-example approaches. It is more efficient than Biermann's, but less robust as the examples must be carefully chosen. A striking difference with Biermann's approach is that recursion is here detected by folding parts of several traces obtained from different examples, whereas Biermann's mechanism detects recursion by folding a single trace onto itself.

The mentioned Basic Synthesis Theorem constitutes a major breakthrough, as it provides a firm theoretical foundation to synthesis from examples. Summers also describes a technique that automatically introduces accumulator parameters when no recurrence relations can be found: this amounts to descending generalization [Deville 90]. The results of Summers have spawned considerable efforts for generalization and improvement, especially by [Jouannaud and Kodratoff 83] [Kodratoff and Jouannaud 84]. Their achievements are very encouraging as the developed sequence matching algorithms are very efficient.

3.3.3 Heuristic Synthesis from Examples

Heuristic synthesis from examples involves more or less heavy use of heuristics in order to prune the search space.

Heuristic synthesis from examples is performed by the systems of [Hardy 75], [Shaw *et al.* 75], and [Siklóssy and Sykes 75], in the sense that they fill in the place-holders of a LISP divide-and-conquer program schema (see Chapter 8) in a plausible way according to the given examples.

A more disciplined usage of heuristics is advocated by [Biermann 84a]: astronomical search-spaces and poorly-understood pruning techniques can be avoided by developing synthesis mechanisms for restricted program classes that are well-understood. Such synthesis mechanisms are very efficient and reliable, and could be components of a large synthesis tool-box. A hierarchical decomposition schema and a production rule mechanism are used by [Biermann and Smith 79] to speed-up the algorithmic synthesis mechanism of [Biermann 78], and this for the so-called "scanning" LISP programs. The resulting mechanism is fairly algorithmic, except for the selection of the used production rules. This approach achieves a neat separation of the synthesis control structure from the programming knowledge that is encoded in the production rules. Efficiency is thus gained at the expense of generality.

Another heuristic schema-based synthesis system from examples is described by [Biggerstaff 84].

3.4 Logic Program Synthesis from Examples

The synthesis of logic programs (Prolog programs, say) is an area of intense research since the early 1980s. Since the inferred concept description is described by a logic program, the examples are presented in the relational form. Stephen Muggleton has recently named this field ILP (*Inductive Logic Programming*) [Muggleton 91], because it is at the intersection of

empirical (inductive) learning and logic programming. The main objectives are (*i*) to upgrade the results of empirical learning (which are largely restricted to the propositional calculus) to the computational (first-order) logic framework of logic programming, and (*ii*) to inject (more) background knowledge into empirical learning from examples.

Unfortunately, there is a potential confusion due to the designation "inductive logic programming". Indeed, one could partition the class of (logic) programs into algorithms (which usually involve some iterating mechanism, such as loops or recursion) and concept descriptions (which do not involve some iterating mechanism). But ILP research seems to focus mostly on concept descriptions, with the hope that the developed learning mechanisms also work for algorithms. However, we believe that the class of all possible logic programs, recursive or non-recursive, is way too large to be efficiently synthesizable by a uniform mechanism. For the purpose of this surveying section, we have to accept this broad view of ILP regarding logic programs, but the remainder of this thesis is only concerned with the synthesis of recursive logic programs.

An introduction to ILP is in preparation [Muggleton 94], and a compilation of the landmark papers is available [Muggleton 92]. Moreover, special ILP workshops exist. This present survey is organized as follows. First, in Section 3.4.1, we present Shapiro's *Model Inference System*, the pioneering learner of ILP. Then, in Section 3.4.2, we briefly discuss other relevant systems.

3.4.1 Shapiro's *Model Inference System*

The foundational work for ILP was laid by the research on subsumption and generalization in the early 1970s by [Plotkin 70, 71] and [Reynolds 70]. Their results, as well as the efforts on extending them, are surveyed in Chapter 7. Then, in the early 1980s, Ehud Y. Shapiro started experimenting with his MIS (*Model Inference System*) as a mechanism for synthesizing Prolog programs from ground (positive and negative) Prolog facts [Shapiro 81]. He later discovered that program synthesis from examples is a particular case of program debugging from test-cases (namely when the initial program is empty), and subsequently called his thesis *Algorithmic Program Debugging* [Shapiro 82]. However, in this discussion, we only focus on the synthesis aspects of MIS. This section is organized as follows. After presenting the synthesis mechanism, we list research on extensions, and finally evaluate MIS.

The Synthesis Mechanism

The (positive and negative) examples are presented one-by-one, and the synthesis mechanism "debugs" its current program accordingly. MIS thus features incremental synthesis.

A program (hypothesis) is a finite set of Horn clauses. The program search space is ordered by a Horn-clause subsumption relation, a particular case of generalization and an extension of Plotkin's subsumption. This allows intelligent pruning of the search space, and thus an improvement over basic identification-by-enumeration:

- *incompleteness*: if a program P fails on some positive example, then no program more specific than P need be considered;
- *incorrectness*: dually, if a program P succeeds on some negative example, then no program more general than P need be considered.

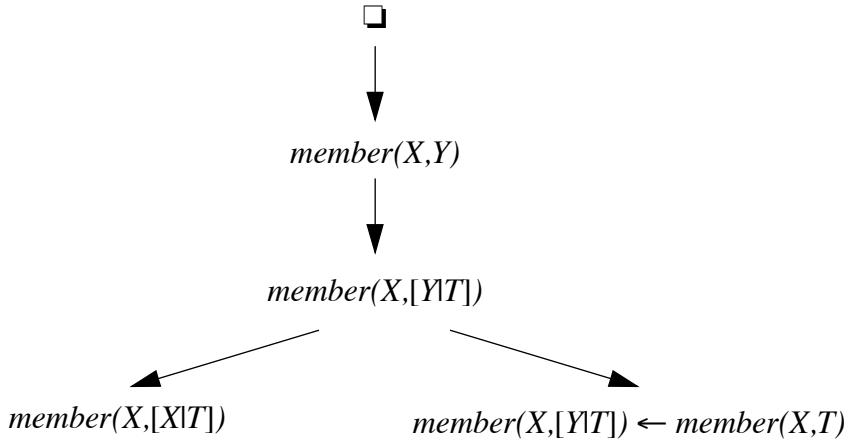
The resulting synthesis mechanism is then as follows (we here omit aspects related to the detection of potential non-termination, which may arise due to the semi-decidability of subsumption checking).

Algorithm 3-1: Incremental synthesis of logic programs from examples.
Input: a (possibly infinite) sequence of positive and negative examples.

Output: a sequence of logic programs covering all the presented positive examples, but none of the presented negative examples.
Algorithm:
set the program *P* to the empty set of clauses;
repeat
  read the next example;
  repeat
    if *P* finitely fails on a known positive example {*P* is incomplete}
      then find an atom *A* that is not covered by *P* and add to *P* a new clause that covers *A*;
    if *P* succeeds on a known negative example {*P* is incorrect}
      then remove from *P* a clause found to be wrong;
  until *P* is complete wrt all positive examples and correct wrt all negative examples;
  write *P*
forever or until no examples are left.

A key feature is the *clause generator* used in case of incompleteness ("bug" correction). It is parameterized on a specialization operator, and thus easily adaptable to different clause languages. We here focus on the operator that enumerates definite Horn clauses, and which is thus used for logic program synthesis. Other operators would enumerate definite clause grammars, and so on. The enumeration of Horn clauses may be carefully designed so as to achieve the objective of intelligent pruning of the search space. The clauses are partially ordered by the subsumption relation into a specialization graph. Thus, if a clause is found to be wrong, then it need never be considered again, and its immediate descendants in the specialization graph are considered plausible replacements. This is an accurate technique for locating and avoiding forbidden features, and definitely a better technique than backtracking. In order to be realistic, this operator however needs a declaration of the basis-set of all the predicates that may appear in clauses. The specialization graph is searched breadth-first.

Example 3-14: Part of the specialization graph for the predicate *member*/2 is as follows (where the vertical arrows denote subsumption):



For convenience, the specialization operator is also given the type information that the first parameter of *member*/2 is a simple term, and that its second parameter is a list. ♦

The "bug" location diagnoses in case of incompleteness and incorrectness are performed by experiment generation and queries to an *oracle*. Shapiro has experimented with three different search strategies to find a clause that covers a given atom (in case of incompleteness):

- the *eager strategy* is the most powerful as it always succeeds, but this is at the expense of many (often superfluous) queries to the oracle;

- the *lazy strategy* is less powerful, but formulates no queries at all, because it relies on the already presented positive examples;
- the *adaptive strategy* is a compromise in power between the first two strategies; it formulates no queries at all, because it relies on the already presented positive examples and on the currently hypothesized program.

The classes of programs that can be obtained via these strategies are different. Identification-in-the-limit is guaranteed by all three strategies under some well-defined pre-conditions. See the evaluation section below for further comments of these strategies.

Let's illustrate all this on a simple synthesis.

Example 3-15: Informally speaking, the *member(E,L)* relation holds iff term E is a member of list L. During the synthesis of a logic program for the *member*/2 predicate, the (here heavily annotated) dialog between MIS and the user goes basically as follows:

MIS: Declare the predicate to be synthesized.
User: *member*/2.
MIS: What are the predicates used by *member*/2?
User: *member*/2. (The user here anticipates that the target logic program is recursive, and that no other predicates will be useful.)
MIS: The initial program P_0 is {}. First example?
User: *member(a,*[*a*]*)*.
MIS: P_0 is incomplete, because *member(a,*[*a*]*)* is not covered. After a call to the clause generator, the revised program P_1 is {*member(X,Y)*}, which is correct wrt the examples presented so far. Next example?
User: ¬ *member(a,*[*b*]*)*.
MIS: P_1 is incorrect, because *member(a,*[*b*]*)* is covered. The first (and only) clause of P_1 is removed. The resulting empty program is incomplete, because *member (a,*[*a*]*)* is not covered. After a call to the clause generator, the revised program is {*member(X,*[*Y*|*T*]*)*}. This program is incorrect, because *member(a,*[*b*]*)* is covered. Its first (and only) clause is removed. The resulting empty program is incomplete, because *member(a,*[*a*]*)* is not covered. After another call to the clause generator, the revised program P_2 is {*member(X,*[*X*|*T*]*)*}, which is correct wrt the examples presented so far. Next example?
User: *member(a,*[*b,a*]*)*.
MIS: P_2 is incomplete, because *member(a,*[*b,a*]*)* is not covered. After a call to the clause generator, the revised program P_3 is {*member(X,*[*X*|*T*]*)*; *member(X,*[*Y*|*T*]*)* ← *member(X,T)*}, which is correct wrt the examples presented so far. Next example?

Suppose the user stops presenting examples. MIS has synthesized a correct logic program for the *member*/2 predicate. This sample synthesis is admittedly a little bit contrived (and actually devoid of queries to the oracle), but the point is here to get a feel for this synthesis mechanism. ♦

Let's summarize: MIS starts from a mixed presentation of multiple, ground, single-predicate, relational, positive and negative examples that are selected by an agent (usually a human specifier) who knows the intended relation, and who is assumed to select only examples that are consistent with the intended relation. The hypotheses are definite Horn clause logic programs. A conceptual bias on the program language is achieved by the declaration of a basis-set of predicates that may appear in the bodies of clauses. The background knowledge consists of logic programs for some of the basis-set predicates. The program space is ordered by the sample-independent subsumption relation, which is however undecidable in general: hence the need for resource-bounded subsumption checking. The used identification criterion is identification-in-the-limit. MIS however doesn't proceed by enumeration, because of

its intelligent organization of the search space. Type, mode, and determinism information about the specified predicate is used for further restriction of the search space. MIS performs incremental (actually even iterative) synthesis, which is moreover breadth-first data-driven, algorithmic, interactive, non-monotonic, consistent, and conservative. Multiple related concepts are learned simultaneously.

Extensions

MIS has spawned a huge variety of efforts for improvement. Some immediate optimizations are proposed by [Huntbach 86].

Other researchers have tackled Shapiro's remarks on oracle mechanization [Shapiro 82, Section 3.7] via the incorporation of "constraints and partial specifications". The solutions by [Lichtenstein and Shapiro 88], [Drabent *et al*. 88], and [De Raedt and Bruynooghe 92] are surveyed in Chapter 6. The solution by [Dershowitz and Lee 87] seems unrealistic in general, because it relies on a complete specification, which could thus rather be used to perform deductive synthesis.

The search for clauses goes from general to specific, using a most general specialization operator to descend a subsumption hierarchy. But subsumption is a weak model of generalization: it is insufficient because many "wanted" generalizations are not obtained, due to the lack of background knowledge, and inadequate because many "unwanted" generalizations are obtained. Background knowledge should thus be used for inferring more "interesting" generalizations. In Chapter 7, we survey the efforts of [Buntine 88] and [Muggleton 91] for introducing stronger models of generalization. The MARKUS system of [Bratko and Grobelnik 93] introduces other improvements of the clause generator.

Finally, the notion of program schema (see Chapter 8) allows a further organization of the search space. The system of [Tinkham 90] is based on the insight that synthesis need not start from the empty program, but could actually start from the most-specific schema that is believed to be applicable. The search space is thus extended to a second-order search space, at the bottom of which are logic programs, and at the top of which are logic program schemas. If synthesis starts from a "good" schema, then the improvement is exponential. A similar approach is taken by the MISST system of [Sterling and Kirschenbaum 91], who develop a new specialization operator, based upon a view of logic programs as skeletons (schemas) to which techniques (standard programming practices) are applied.

Evaluation

The complexity and power of MIS compare favorably with the algorithmic synthesis mechanisms of Biermann and Summers (see Section 3.3.2). Since it is parameterized on a specialization operator, it is much more adaptable than these other mechanisms. However, MIS is much less data-efficient than the latter.

The eager search strategy is independent of the order of presentation of the examples. However, the lazy search strategy is order-dependent. [Shapiro 82, page 110] shows how an (inadvertent) adversary user might con MIS (equipped with the lazy strategy) into synthesizing an arbitrarily large program: the example sequence *member(a,*[*a*]*)*, *member(b,*[*a,b*]*)*, *member(c,*[*a,b,c*]*)*,…, generates the logic program {*member(X,*[*X*|*T*]*)*; *member(X,*[*Y,X*|*T*]*)*; *member(X,*[*Z,Y,X*|*T*]*)*; …}. This is clearly not a desirable behavior. But the research of [Jantke 89] [Jantke and Goldammer 91] seems a possible solution to this situation, as their synthesis mechanism starts from precisely such clause sequences.

The adaptive strategy is less order-dependent, and said to be the most useful one in practice. However, it may result in "dead code" (clauses that always lead to failure), namely when it relies on a clause yet unknown to be incorrect. Again, this is not a desirable behavior.

Another consequence of order-dependence arises because MIS synthesizes Prolog programs, rather than some form of logic algorithms: an (inadvertent) adversary user might con MIS into synthesizing the base-case clause after the recursive clause. While declaratively correct, such a program goes into infinite loops for certain modes.

Moreover, as reported by [Sterling and Kirschenbaum 91], MIS is likely to generate redundant clauses. Removing these seems too computationally expensive to be practical, though the work of [Buntine 88] gives promising ideas towards this.

All the predicates used by the clause generator must be declared prior to synthesis, and an oracle is assumed to be willing to answer questions about them. Multiple related concepts may thus be learned simultaneously, but this nice feature doesn't prevent MIS from being unable to invent its own predicates, not to mention implementing them.

Other weak spots are those addressed by the above-mentioned extensions. However, MIS should not be discredited, because it is after all a pioneering, ground-breaking system.

3.4.2 Other Systems

The other ancestor system of ILP, though much less influential, is CONFUCIUS (see [Cohen and Sammut 84]), which has been very much inspired by Banerji's research (see [Banerji 84]). From positive and negative examples, this system incrementally synthesizes a program by oscillating between over- and under-generalization. The mechanism is however largely heuristic-based. It has no background knowledge and no oracle. Recursion is discovered in a way similar to the trace-folding of [Biermann 78]. A successor system is MARVIN [Sammut and Banerji 86], which also uses only a single generalization rule in its specific-to-general search. But it uses background knowledge, and asks questions to an oracle in order to test its hypotheses. It is unable to invent its own predicates, but rather expects to be taught simple concepts before complex ones.

The INDUCE systems (see [Michalski 84]) were among the first to address the issues of using constructive rules of inductive inference. The learning mechanism is a bottom-up approximation-driven one, though heavily based on heuristics.

The CIGOL ("logic" backwards) system of [Muggleton and Buntine 88] performs interactive, incremental synthesis from a mixed presentation of multiple, ground, relational, positive and negative examples, in the presence of background knowledge. It also addresses constructive induction. The technique is based on the ingenious insight that if resolution of clauses (by unification of two literals of the same predicate p, but of opposite signs) may make disappear predicate p in the resolvent clause (namely if there are no other occurrences of p in the two resolved clauses), then inverse resolution (using anti-unification) may make appear some predicate, and its clausal definition. The system is restricted to generating unit clauses for the invented predicate, and the teacher is then asked to name the latter from these unit clauses (which are nothing else but non-ground examples). If the identified name is unknown to the system, then synthesis occurs from these unit clauses. The successor system, GOLEM [Muggleton and Feng 90], extends these ideas. Another descendant system is ITOU [Rouveirol 91], which is based on the inversion of logical entailment, rather than on the inversion of resolution.

The CLINT (*Concept Learning in an INTeractive way*) system of [De Raedt and Bruynooghe 88, 89, 90, 92] incrementally learns multiple related concepts simultaneously, from a mixed presentation of multiple, ground, relational, positive and negative examples, in the presence of background knowledge and integrity constraints. The "bug" correction component is an adaptation of MIS. Identification-in-the-limit is guaranteed. The system automatically shifts its syntactic bias when needed. It generates itself almost all the examples it needs, and interacts with an oracle for the classification of these examples as positive or neg-

ative ones. It assimilates newly learned concept by adding them to its knowledge base. Predicate invention is tackled by analogical inference wrt assimilated concepts.

The FOIL (*First-Order Inductive Learner*) system of [Quinlan 90] extends the propositional attribute-value learners to the first-order formalism of relations and Horn clauses, in view of benefitting from the additional expressive power. The system is very different from all others described here, in that it performs passive, non-incremental learning from positive and negative examples. Compared to MIS, it constructs clauses literal-by-literal, rather than using a clause-generator; the notions of proof, oracle, and background knowledge are absent; and the hypothesized program is the maximally-general one, rather than the maximally-specific one, as in MIS. On the negative side, its greedy, heuristic-based search for suitable literals to expand a clause prevents all backtracking. Recursive programs are considered, though at the expense of potential non-termination. There is no predicate invention. Moreover, extremely large example sets are needed for successful synthesis: for instance, 10,261 examples are needed for *append*/3, and 4,761 examples are needed for *reverse*/2. All this shows that FOIL seems inadequate for recursive program synthesis.

The LINUS system of [Lavrač and Džeroski 92] learns relational concept descriptions, but works by first transforming the examples into propositional form, and then using propositional attribute-value learners (which may handle noisy examples).

A completely different school of thought is advocated by [Hagiya 90]. He re-formulates Summers' recurrence relation detection mechanism in a logic framework, using higher-order unification in a type theory with a recursion operator. The method is extended to synthesizing a deductive proof-by-induction from a concrete sample proof.

3.5 Conclusions on Program Synthesis from Examples

In Section 3.2.4, we have defined algorithm synthesis from examples as a niche of empirical learning from examples. As a reminder, for algorithm synthesis, we are here only interested in the setting with human specifiers who know (even if only informally) the intended relation, and who are assumed to choose only consistent examples. Moreover, the intended relation is assumed to have a recursive algorithm. There is a general consensus that synthesis from examples would be a useful component of any larger synthesis system. So we now draw some conclusions about the approaches surveyed in the previous two sections.

The surveyed mechanisms of LISP program synthesis actually turn out to be pretty much orthogonal to those of Prolog program synthesis. They were developed in different time-periods (until the late 1970s, and since the early 1980s, respectively), and the latter seem to have completely supplanted the former. This supersession is largely due to the "declaration of defeat" by the former (because of the mechanisms, not because of the target language), and to the ensuing hunger for new approaches. The growing awareness of the early 1980s that concept descriptions and logic programs may share the same formalism has even provoked a gradual absorption of program synthesis from examples by the field of empirical learning. This culminates in the designation "*Inductive Logic Programming*", which is unfortunately a misleading name because it gives too much tribute to "programming", compared at least to the current trends of ILP research. The logic programming paradigm has blurred the distinction between learning-from-examples and programming-from-examples. While academically rewarding, this new perception seems very dangerous to us in practice: as Section 3.2.4 shows, the superclass relation between both fields is sufficiently "strict" to justify highly specialized mechanisms for the latter. At least now, where both fields are still young, though maturing. The future will tell whether there effectively is a unique and efficient learning mechanism for all classes of concepts. However, we strongly doubt this!

All the surveyed mechanisms perform pretty well on predicates that involve only structural manipulations of their parameters, such as list concatenation or list reversal. But as soon as the actual values of the parameters are relevant (such as for list sorting), problems arise due to the inherent incompleteness and ambiguity of specifications by examples. There is thus an important need for making more knowledge available to synthesis, because such knowledge is after all available to human algorithm designers. Knowledge may be given under the form of extended incomplete specifications, of background knowledge (domain knowledge), and of algorithm design knowledge. Whereas (some of) the 1970s approaches were pretty much on the analytic (EBL) side of machine learning (though without background knowledge, and thus necessarily enumerative), most of the 1980s-and-beyond approaches are definitely on the empirical side of machine learning. Today's trend seems fortunately to be towards an integration of analytic (knowledge-reliant) and empirical (user-reliant) learning, and this in view of cross-fertilization for overcoming drawbacks.

Although human learning is naturally incremental and non-monotonic, we believe that automated algorithm synthesis from examples should be non-incremental and monotonic. Indeed, incremental synthesizers tend to be very "undisciplined" due to their non-monotonic debugging (patchwork) approach. This need not always be the case, for instance if the examples are "well-chosen". But one should bear in mind that the specifier is not always supervising the synthesis. And even if s/he were, putting the synthesizer (back) on the right track with a carefully crafted example goes counter the most frequent setting where the specifier doesn't know how to best define the intended relation by an algorithm, and is possibly even unable to understand an algorithm. The risks of infinite, redundant, or dead code due to the potential dependence on the order of the presented examples are symptomatic for some incremental mechanisms that have no understanding of what they are synthesizing. Considering that an example embodies several kinds of information, incremental synthesis has to extract all these kinds of information from the examples presented so far, whereas non-incremental synthesis may extract only one kind of information from every example at each step. The latter seems to be a much more reasonable approach. We thus believe that a synthesis mechanism should carefully design, just like an expert human designer (and unlike an inexpert human learner), a correct algorithm in a monotonic way from a fixed set of given examples.

The current trend of ILP research seems [Muggleton 91] to be towards handling noisy examples, and to PAC-learning, whose probabilistic approach to correctness is however inherently incompatible with the goals of algorithm synthesis. The most promising applications of ILP research seem [Muggleton 91] to be when the intended concept is unknown, which also goes counter our algorithm synthesis setting. There is general skepticism about the usefulness of schemas, because these tend either to spell out the entire search space (which is fruitless), or to be too domain-specific (and hence difficult to obtain), if they exist at all. All this clearly indicates a desire to do empirical learning in general, rather than "only" algorithm synthesis. The achievements of ILP are already very impressive [Muggleton 91]. But this quest for a general mechanism is inevitably at the cost of a poor synthesis of recursive algorithms, because the developed learners often merely perform some kind of example-analysis in a pre-enumerated search space, whereas algorithm synthesis requires more complex operations such as generating logic quantifiers. We are here not interested in learning to recognize Michalski's trains, or Winston's arches.

4 A Logic Program Development Methodology

This research is set in the framework of the logic program development methodology described in [Deville 90]. In order to keep this thesis self-sufficient, we summarize that methodology, and put special focus on aspects directly related to our research. This summary mainly states results, and not always their motivations or proofs: refer to [Deville 90] when in need of further explanations.

The original promise of programming in (first-order) logic when using Prolog is impaired by the following facts: [5]

- the Prolog inference engine is incomplete, unfair, and unsound;
- the Prolog inference engine approximates logical negation by negation-as-failure;
- the Prolog language embodies predicates such as:
  - control predicates (*cut*/0, *fail*/0,…),
  - extra-logical predicates (input/output predicates, *clause*/2, *assert*/1, *retract*/1,…),
  - meta-logical predicates (*ground*/1, *var*/1, *call*/1, ==/2,…),
  - second-order predicates (*setof*/3, *bagof*/3,…),

  which allow programming outside first-order logic;
- Prolog programs are rarely multi-directional.

But all these features have been deliberately chosen so as to make Prolog a practical programming language.

So there is a need for a maximally language-independent logic programming methodology that reconciles this gap between the declarative and the procedural semantics. Alternative approaches would have been either to forget about declarative logic programming, or to design a new language not suffering from that gap, or to design a new logic filling that gap. But these approaches have been judged not to be very attractive.

Such a methodology has been formulated. It aims at programming-in-the-small, and is (mainly) meant for algorithmic problems. It is sub-divided into three stages: [6]

- Stage A: Elaboration of a specification;
- Stage B: Design of a logic algorithm (and possibly its transformation);
- Stage C: Derivation of a logic program (and possibly its transformation).

Stage B is based only on the declarative semantics of logic, and is independent of the target logic programming language used at Stage C.

An integrated logic programming environment, called *Folon*,[7] is being developed to support the entire methodology [Henrard and Le Charlier 92].

Note however that this thesis is independent of that methodology. We just use its framework because of its useful notations and its focus on the logical aspect of logic program development.

This chapter is organized as follows. In Section 4.1, we discuss the elaboration of specifications. Section 4.2 is about the design of logic algorithms, whereas Section 4.3 is about their transformation. Section 4.4 describes the derivation of logic programs, whereas Section 4.5 describes their transformation.

5. This chapter is written as if there (already) were a standard Prolog.
6. Note that we say “logic algorithm” where [Deville 90] says “logic description”. This terminology was already used in [Deville 87], which was the starting point for [Deville 90].
7. The environment is named after Jean-Michel Folon, a famous contemporary Belgian painter.

```
Procedure
    compress(L,C)
Types
    L: list of Term
    C: list of <Term,Integer>
Restrictions on Parameters
    compactList(C)
Directionalities
    in(ground,novar)::out(ground,ground) <0-1>
    in(ground, var )::out(ground,ground) <1-1>
Relation
    C is the compression of L
```

Figure 4-1: *Spec(compress)*

4.1 Elaboration of a Specification

The first stage of the methodology is the elaboration of a specification. Informally, a specification of a procedure is a statement describing the problem that is to be implemented, as well as how to correctly use the resulting procedure. We here restrict the focus to side-effect-free problems.

Definition 4-1: A *specification* of a procedure for predicate *r*/*n*, denoted *Spec(r)*, is composed of a procedure declaration, a type declaration for each parameter, a set of restrictions on parameters, a set of directionalities, and a relation definition.

We present the involved components only by means of a few sample specifications.

Example 4-1: The *compress(L,C)* procedure succeeds iff *C* is a compact list of $<v_i,c_i>$ couples, such that the i^{th} plateau of list *L* has c_i elements equal to v_i. Hence the sample specification of Figure 4-1. Note that the first four entries are here written in a formal language, while the relation entry is non-formal. The first three entries are assumed to be self-explanatory. The first directionality says that given *L* ground and *C* a non-variable, there is at most one ground instance of *C* such that *compress(L,C)* succeeds. The second directionality says that given *L* ground and *C* a variable, there is exactly one ground instance of *C* such that *compress(L,C)* succeeds. Other directionalities could have been given (for instance for *L* non-ground), but we only want a procedure satisfying the two given ones.

Example 4-2: The *firstPlateau(L,P,S)* procedure succeeds iff *P* is the first maximal plateau of the non-empty list *L*, and list *S* is the corresponding suffix of *L*. A sample specification is given in Figure 4-2.

The different entries of a specification act as pre-conditions, or as post-conditions, or as both.

Definition 4-2: The *domain* of a procedure for predicate *r*, denoted *dom(r)*, is the intersection of the Cartesian product of the types of its parameters and the relation defined by the restrictions on the parameters.

Definition 4-3: The *specified relation* is the intersection of the domain and the relation defined in the specification.

The specified relation is assumed to be identical to the *intended relation*, which is denoted $\mathcal{R}$. Intuitively speaking, a procedure that correctly implements a problem computes the intersection of its domain and its specified relation, and "satisfies" its directionalities.

```
Procedure
    firstPlateau(L,P,S)
Types
    L,P,S: list of Term
Restrictions on Parameters
    L≠[] ∧ P≠[]
Directionalities
    in(ground,novar,novar)::out(ground,ground,ground) <0-1>
    in(ground, var , var )::out(ground,ground,ground) <1-1>
Relation
    P is the first maximal plateau of L,
     and S is the corresponding suffix of L
```

Figure 4-2: *Spec(firstPlateau)*

$$
\begin{array}{l}
\texttt{compress(L,C)} \Leftrightarrow \\
\quad \texttt{L=[]} \qquad\qquad\qquad \wedge\ \texttt{C=[]} \\
\vee\ \texttt{L=[HL]} \qquad\qquad\ \wedge\ \texttt{C=[<HL,1>]} \\
\vee\ \texttt{L=[}HL_1\texttt{,}HL_2\texttt{|TL]} \quad \wedge\ HL_1 \neq HL_2 \\
\qquad\qquad\qquad\qquad\ \wedge\ \texttt{compress([}HL_2\texttt{|TL],TC)} \\
\qquad\qquad\qquad\qquad\ \wedge\ \texttt{C=[<}HL_1\texttt{,1>|TC]} \\
\vee\ \texttt{L=[}HL_1\texttt{,}HL_2\texttt{|TL]} \quad \wedge\ HL_1 = HL_2 \\
\qquad\qquad\qquad\qquad\ \wedge\ \texttt{compress([}HL_2\texttt{|TL],TC)} \\
\qquad\qquad\qquad\qquad\ \wedge\ \texttt{C=[<V,s(N)>|TTC]} \wedge\ \texttt{TC=[<V,N>|TTC]}
\end{array}
$$

Logic Algorithm 4-1: *LA(compress)*

4.2 Design of a Logic Algorithm

The second stage of the methodology is the design of an algorithm, independently of any logic programming language, starting from the specification of a procedure. Such an algorithm is expressed in first-order logic, and called a logic algorithm. The design is based solely on the declarative semantics of first-order logic. This implies that the directionality information from the specification is not used at this stage.

Definition 4-4: A *logic algorithm* defining a predicate *r/n*, denoted *LA(r)*, is a closed well-formed formula of the form:

$$\forall X_1 \ldots \forall X_n \quad r(X_1,\ldots,X_n) \Leftrightarrow F$$

where the X_i are distinct variables, and F is a well-formed formula. The atom $r(X_1,\ldots,X_n)$ is called the *head*, and F is called the *body* of the logic algorithm.

In the sequel, we drop the universal quantifications in front of the heads, as well as any existential quantifications at the beginnings of bodies of logic algorithms.

Example 4-3: A sample version of *LA(compress)* is Logic Algorithm 4-1.

Example 4-4: A sample version of *LA(firstPlateau)* is Logic Algorithm 4-2.

Note that logic algorithms take the "natural" form of what one would expect to be a logic program (that implements a "programming problem"). Indeed, the equivalence symbol is necessary to state when *r/n* is *true*, as well as when *r/n* is *false*. Logic algorithms correspond in fact to the notion of completed logic programs [Clark 78]. Hence, reasoning on logic algorithms is equivalent to using logic programs, but reasoning on their completions.

Deville introduces the following important differences with "classical" approaches to logic programming:

```
firstPlateau(L,P,S) ⇔
    L=[HL]                 ∧ P=L ∧ S=[]
  ∨ L=[HL1,HL2|TL]         ∧ HL1≠HL2
                           ∧ P=[HL1] ∧ S=[HL2|TL] ∧ list(TL)
  ∨ L=[HL1,HL2|TL]         ∧ HL1=HL2
                           ∧ firstPlateau([HL2|TL],TP,TS)
                           ∧ P=[HL1|TP] ∧ S=TS
```

Logic Algorithm 4-2: *LA(firstPlateau)*

- a first-order language is defined independently of any first-order theories: let $\mathcal{F}$ and $\mathcal{Q}$ be disjoint sets of functors and predicate symbols, respectively, and $\mathcal{V}$ be the set of variable symbols; $\mathcal{T}$ is the set of all terms that can be formed from $\mathcal{F}$ and $\mathcal{V}$; the Herbrand universe $\mathcal{U}$ is the set of all ground terms that can be formed from $\mathcal{F}$; $\mathcal{A}$ is the set of all atoms that can be formed from $\mathcal{Q}$ and $\mathcal{T}$; the Herbrand base $\mathcal{B}$ is the set of all ground atoms that can be formed from $\mathcal{Q}$ and $\mathcal{U}$; $\mathcal{W}$ is the first-order language, that is the set of all well-formed formulas that can be formed from $\mathcal{A}$, $\mathcal{V}$, and the quantifiers;
- the focus is restricted to Herbrand interpretations, Herbrand models, Herbrand-logical consequences, and Herbrand-satisfiability.

The correctness of a logic algorithm wrt its specification is an important issue. All the predicates used in a logic algorithm are considered as primitives, and the existence of correct logic algorithms is thus assumed for all of them. Throughout this thesis, we adopt at least the following set of primitives: $\{=/2, \neq/2, </2, >/2, \leq/2, \geq/2, \mathit{list}/1, \mathit{odd}/1, \mathit{true}/0, \mathit{false}/0\}$.

This allows a definition of logic algorithm correctness that is only in terms of the specifications of the used predicates. The definition establishes an equivalence between the specified relation and the set of Herbrand-logical consequences of the logic algorithm.

Definition 4-5: Let *Spec(r)* be a specification, *LA(r)* be a logic algorithm, and $\mathcal{A}$ be a finite set of logic algorithms containing *LA(r)*. Then *LA(r)* is *(totally) correct* in $\mathcal{A}$ wrt *Spec(r)* iff, for any ground *n*-tuple $\boldsymbol{t}$, the following two conditions hold:

$$\mathcal{A} \models r(\boldsymbol{t}) \quad \text{iff} \quad \boldsymbol{t} \in \mathcal{R} \text{ and } \boldsymbol{t} \in dom(r)$$

$$\mathcal{A} \models \neg\, r(\boldsymbol{t}) \quad \text{iff} \quad \boldsymbol{t} \notin \mathcal{R} \text{ or } \boldsymbol{t} \notin dom(r)$$

The second condition is necessary because negation can be used in logic algorithms. Note that this definition is restricted to ground terms: this simplifies proofs, and does not affect the existence of correct witnesses to existential queries. Also note that this definition implies that a logic algorithm has to verify whether its parameters belong to the domain of the procedure. The opposite approach is taken at the logic program level.

A correct logic algorithm is Herbrand-satisfiable. The truth value of a ground atom *r(t)* is the same in every Herbrand model of a correct logic algorithm *LA(r)*. A correct logic algorithm remains correct if a new logic algorithm is added.

There are of course many methodologies of logic algorithm design. Deville discusses three of them, namely:

- construction by structural induction;
- top-down decomposition;
- iteration through universal quantification.

Let's have a look at these methodologies, in Section 4.2.1 to Section 4.2.3, respectively.

4.2.1 Construction by Structural Induction

The most important logic algorithm design methodology is based on structural induction. The principle of well-founded induction suggests constructing a logic algorithm by induction

over the structure of some parameter. The value of that *induction parameter* is reduced to something smaller according to some well-founded relation, and a partial result is recursively computed. If no reduction is possible, then the problem is solved directly. There are four steps to this design methodology:

- Step 1: Selection of an induction parameter;
- Step 2: Selection of a well-founded relation;
- Step 3: Selection of the structural forms of the induction parameter;
- Step 4: Construction of the structural cases.

Let's explain these four steps one by one. Note that this method is proven to yield correct logic algorithms. A tool, called *Logist*, is being developed for the *Folon* environment to assist a designer in following these steps [Burnay and Deville 89].

Step 1: *Selection of an Induction Parameter*

The first step is the selection of an induction parameter. Only parameters of inductive types (such as integers, lists, trees, sets, strings,…) are eligible as induction parameters. For simplicity, throughout this thesis, we assume that no parameters are tuples.

Definition 4-6: A *simple induction parameter* is composed of one parameter.
A *compound induction parameter* is composed of at least two parameters.

Most problems can be implemented by logic algorithms with induction over a simple induction parameter.

The selection of a suitable induction parameter is straightforward, but two useful heuristics have been identified:

Heuristic 4-1: *Functionality Heuristic*
Select an induction parameter such that, given a ground instance of it, the specified relation holds for at most one ground instance of the other parameters.

This heuristic usually simplifies the remainder of the design process.

Heuristic 4-2: *Directionality Heuristic*
Select an induction parameter that is ground in all the *in*-parts of the given directionalities.

This heuristic only reveals its advantages when one looks ahead: at the logic program derivation stage (Section 4.4), domain checking and termination proofs are simplified, and the derived logic programs usually are more efficient. One could thus argue that this heuristic is the best.

Example 4-5: For the *compress* predicate, both heuristics guide towards selecting *L* as induction parameter.

In the sequel, when talking about a logic algorithm *LA(r)*, we sometimes write *LA(r-X)* to show that *X* was selected as induction parameter. Note however that the predicate defined by *LA(r-X)* still is *r*, and not *r-X*.

Step 2: *Selection of a Well-Founded Relation*

The second step is the selection of a well-founded relation over the type of the induction parameter. Two heuristics for the selection of a well-founded relation have been identified:

Heuristic 4-3: *Intrinsic Heuristic*
Select a well-founded relation reflecting the definition of the type of the induction parameter.

This heuristic is easy to apply, since the definition itself of an inductive type already suggests a well-founded relation.

Example 4-6: For the *compress* predicate, the intrinsic heuristic guides towards selecting "is the tail of" as well-founded relation over the type of *L*. Weaker variants such as "has less elements than" or "is a suffix of" are also suitable, but it is best to start with strong relations, and relax them as needed.

Heuristic 4-4: *Extrinsic Heuristic*
Select a well-founded relation that reflects the definition of the type of some other parameter, or that reflects the structure of the specified relation.

Applying this heuristic is often more complicated, but the ensuing construction steps can often be simplified.

Example 4-7: For the *compress* predicate, the extrinsic heuristic guides towards selecting "has one plateau less than" as well-founded relation over the type of *L*, because it reflects the structure of the *C* parameter. But we retain the well-founded relation suggested by the Intrinsic Heuristic for the remainder of this section, and come back to this decision in Section 5.2.2.

None of these two heuristics is superior to the other. In the sequel, when talking about a logic algorithm *LA(r)*, we sometimes write *LA(r-int-X)*, or *LA(r-ext-X)*, to show that *X* was selected as induction parameter, and that the intrinsic (respectively extrinsic) heuristic was applied. Again, the predicate defined by *LA(r-int-X)* still is *r*, and not *r-int-X*.

An interesting exercise is to compare logic algorithms designed by induction on different parameters, or using different well-founded relations. Considering the heuristics above, it is no surprise that, for a binary predicate *r* having *X* and *Y* as parameters, *LA(r-int-X)* and *LA(r-ext-Y)* are structurally similar, or that *LA(r-int-Y)* and *LA(r-ext-X)* are structurally similar.

Step 3: *Selection of the Structural Forms of the Induction Parameter*

The third step is the selection of the structural forms of the induction parameter. *Structural forms* are terms. This step consists of finding at least one minimal form and at least one non-minimal form for the induction parameter, such that they are all mutually exclusive over the domain of the induction parameter. A *non-minimal form* is a structure case of the domain of the induction parameter. A *minimal form* is a base case of the domain of the induction parameter, that is a form to which at least one sample computation starting from a non-minimal form eventually reduces the induction parameter.

Example 4-8: For the *compress* predicate, we select [] as the minimal form, and $[HL_1|TL]$ as the non-minimal form of the induction parameter *L*.

Step 4: *Construction of the Structural Cases*

The fourth and last step is the construction of the structural cases. This amounts to formalizing how the other parameters relate to the induction parameter, for each of its structural forms. The results are *structural cases*. A structural case is a *minimal case* if the induction parameter is of a minimal form, and a *non-minimal case* if the induction parameter is of a non-minimal form. This step is the most creative one, and requires a lot of skill. It is actually divided into two sub-steps:

- Step 4-1: Construction of the structural cases, independently of domain membership;
- Step 4-2: Introduction of domain-checking literals.

The second sub-step is important in view of achieving correctness of the designed logic algorithm. Indeed, the logic algorithm must evaluate to *false* in case the parameters do not belong to the domain. With a domains-as-preconditions approach, domain checking literals

```
compress(L,C) ⇔
    L=[]        ∧ C=[]
  ∨ L=[HL1|TL]∧ TL=[]
              ∧ C=[<HL1,1>]
  ∨ L=[HL1|TL]∧ TL=[HL2|TTL] ∧ HL1=HL2
              ∧ compress(TL,[<HL1,N>|TC])
              ∧ C=[<HL1,s(N)>|TC]
  ∨ L=[HL1|TL]∧ TL=[HL2|TTL] ∧ HL1≠HL2
              ∧ compress(TL,TC)
              ∧ C=[<HL1,1>|TC]
```

Logic Algorithm 4-3: *LA(compress-int-L)*

would have to be added to ensure that the used predicates are correctly used. Note however that in most situations, Step 4-2 reduces to doing nothing at all.

Example 4-9: For the *compress* predicate, we proceed as follows:

- if *L* is of the selected minimal form (*L*=[]), then *C* must be empty, too (*C*=[]);
- if *L* is of the selected non-minimal form (*L*=[HL_1|*TL*]), then:
  - either *TL* is empty, and *C* must then be [<HL_1,1>];
  - or *TL* is non-empty and starts with a term identical to HL_1, and *C* must then be [<HL_1,*s(N)*>|*TC*], where *compress(TL,*[<HL_1,*N*>|*TC*]*)* holds;
  - or *TL* is non-empty and starts with a term different from HL_1, and *C* must then be [<HL_1,1>|*TC*], where *compress(TL,TC)* holds.

No domain-checking literals need to be added. We obtain Logic Algorithm 4-3, which is equivalent to Logic Algorithm 4-1. ♦

Generalization of a Problem

Sometimes, it turns out to be difficult—if not impossible—at Step 4 to reduce a problem into a sub-problem that can be solved with a recursive use of the logic algorithm under design. Besides questioning earlier decisions made at Steps 1 to 3, one can also generalize the problem, so that the sub-problem and the original problem are both special cases of the general one. Paradoxically, such a general problem often turns out to be easier to implement. A logic algorithm is designed for the generalized problem, and a logic algorithm for the original problem is expressed using the general one. Generalized specifications often lead to more efficient logic programs, because the underlying logic algorithms incorporate some form of loop merging. Two interesting generalization strategies are:

- structural generalization: generalization of the type of a parameter;
- computational generalization: generalization of a state of computation.

But we do not discuss them here.

4.2.2 Top-Down Decomposition

A logic algorithm can also be designed by decomposition of the original problem into a conjunction of simpler sub-problems. This is a top-down methodology if the sub-problems have not yet been implemented. Otherwise, it's a bottom-up methodology, because of the re-use of previously implemented sub-problems. The resulting logic algorithms are usually non-recursive.

Example 4-10: The *append3(A,B,C,ABC)* procedure succeeds iff list *ABC* is the concatenation of lists *A*, *B*, and *C*. This problem can be decomposed into concatenating *B* to the end of *A*, yielding *AB*, and then concatenating *C* to the end of *AB*, yielding *ABC*. Hence:

```
append3(A,B,C,ABC) ⇔ append(A,B,AB) ∧ append(AB,C,ABC)
```

where *append(X,Y,Z)* holds iff list *Z* is the concatenation of list *Y* to the end of list *X*. ♦

4.2.3 Iteration through Universal Quantification

A logic algorithm can also be designed by iteration through universal quantification. This requires full first-order logic in the bodies of logic algorithms. The methodology is also known as *iteration through negation*, because the derived logic program performs an iteration by means of negation-as-failure and backtracking.

Example 4-11: The *minimum(L,M)* procedure succeeds iff integer *M* is the minimum element of non-empty integer list *L*. This problem can be implemented as follows:

```
minimum(L,M) ⇔ member(M,L) ∧ ∀E (member(E,L) ⇒ M≤E)
```

where *member(E,L)* holds iff *E* is an element of list *L*. ♦

4.3 Transformation of a Logic Algorithm

Correctness-preserving transformations of logic algorithms aim at allowing the derivation of a more efficient logic program at Stage C. This approach favors the design of a correct logic algorithm, and subsequent transformation thereof into an equivalent logic algorithm. The truly creative effort thus goes into the design of a first, correct logic algorithm, without any concern for its form and complexity. Moreover, the ensuing transformation is often easier to perform than the design from scratch of an optimized logic algorithm.

A set of correctness-preserving transformation rules (such as folding, unfolding,…) has been developed, but we don't list them here. These rules allow:

- the syntactic simplification of logic algorithms;
- the transformation of a logic algorithm into another one that reflects induction on a different parameter;
- the transformation of a logic algorithm designed without induction into another one that does reflect induction on some parameter;
- the transformation of a logic algorithm into another one that reflects a computational generalization of the original problem;
- the merging of several predicates sharing some parameter(s) into a single predicate;

and so on. These transformations only deal with the declarative semantics of logic.

4.4 Derivation of a Logic Program

The third and last stage of the methodology is the derivation of a logic program, starting from the designed logic algorithm. This is a language-specific task, and has thus to take into account all aspects that make logic programming different from programming in a specific language (Prolog, say). These aspects are the features of the underlying inference engine (such as incompleteness, unsoundness, and unfairness due to the search rule, the computation rule, the negation-as-failure rule, and the absence of occur-check), as well as some primitives of the programming language (extra-logical, meta-logical, second-order, and control predicates). In other words, the procedural semantics of logic is now considered, and the directionality information of specifications is thus used.

Definition 4-7: A *pure logic procedure* of a predicate *r*/*n* is a finite sequence of program clauses of predicate *r*/*n*, each with the same *n* distinct variables in its head.

Pure logic procedures are also known as normalized logic procedures.

Definition 4-8: A *(pure) logic program* of a predicate *r/n*, denoted *LP(r)*, is a finite set of (pure) logic procedures of predicate *r/n*.

Definition 4-9: A *(pure) Prolog program* of a predicate *r/n*, also denoted *LP(r)*, is a (pure) logic program of predicate *r/n*, possibly augmented with Prolog control predicates.

The correctness of a logic procedure wrt its specification is an important issue. All the predicates used in a logic procedure are considered as primitives, and the existence of correct logic procedures is assumed for them. This allows a definition of logic procedure correctness that is only in terms of the specifications of the used predicates. [8]

Definition 4-10: Let *Spec(r)* be a specification, *LP(r)* be a logic procedure, and $\mathcal{P}$ be a finite logic program containing *LP(r)*. Then *LP(r)* is *correct* in $\mathcal{P}$ wrt *Spec(r)* iff, for any *n*-tuple ***t*** such that:

- ***t*** is "compatible" with *dom(r)*;
- ***t*** "satisfies" the *in*-part of a directionality $in(m_1,\ldots,m_n)::out(M_1,\ldots,M_n)$ <*min-max*>;
- *Subst* is the sequence of answer substitutions for $\mathcal{P} \cup \{\leftarrow r(\boldsymbol{t})\}$ (using some SLDNF refutation procedure);

the following four conditions hold:

- *Subst* is "partially correct", "complete", and "minimal" for *r(**t**)*;
- if *Subst* is finite, then $\mathcal{P} \cup \{\leftarrow r(\boldsymbol{t})\}$ terminates;
- $\forall \sigma \in Subst$: $\boldsymbol{t}\sigma$ "satisfies" $out(M_1,\ldots,M_n)$;
- *Subst* "satisfies" the multiplicity <*min-max*>.

Note that this definition implies that a logic procedure need not verify whether its parameters are compatible with the domain, because this is actually a pre-condition now. The opposite approach is taken at the logic algorithm level.

Any SLDNF refutation procedure is complete for correct logic procedures. A correct logic program remains correct if a new logic procedure is added.

A straightforward syntactic translation of a logic algorithm into a pure logic program already achieves partial correctness and completeness wrt the specification, whatever the used SLDNF refutation procedure. This translation amounts to a "de-completion" process, as the completion of the pure logic program resulting from the syntactic translation of a logic algorithm is logically equivalent to that logic algorithm.

The remaining, rather technical correctness criteria are specific to every SLDNF refutation procedure. Some aspects to be verified are:

- is the computation rule safe for every goal?
- are the domain and directionality pre/post-conditions satisfied for each subgoal?
- if required, is termination achieved?
- is the sequence of answer substitutions minimal?, and does it satisfy the multiplicity?
- does the absence of occur-check affect soundness?

These aspects are usually a simple manual task. They can also be handled by data-flow analysis or abstract interpretation techniques that reveal, for each directionality, a correct permutation of the program clauses, and a correct permutation of the literals in the body of each clause, and sometimes the addition or deletion of literals. Multi-directionality is usually difficult to achieve. The corresponding tool of the *Folon* environment is described in [De Boeck and Le Charlier 90].

Example 4-12: For the *compress* predicate, the pure logic program derived from Logic Algorithm 4-3 is given in Figure 4-3.

8. This definition involves here undefined concepts: we assume the reader intuitively grasps their meanings, which are of course defined in [Deville 90].

```
compress(L,C) ← L=[],
                C=[]
compress(L,C) ← L=[HL1|TL],TL=[],
                C=[<HL1,1>]
compress(L,C) ← L=[HL1|TL],TL=[HL2|TTL],
                HL1=HL2,
                compress(TL,[<HL1,N>|TC]),
                C=[<HL1,s(N)>|TC]
compress(L,C) ← L=[HL1|TL],TL=[HL2|TTL],
                not(HL1=HL2),
                compress(TL,TC),
                C=[<HL1,1>|TC]
```

Figure 4-3: *LP(compress-int-L)*

```
compress([],[])
compress([HL1],[<HL1,1>])
compress([HL1,HL2|TTL],[<HL1,s(N)>|TC]) ←
                HL1=HL2,!,
                compress([HL2|TTL],[<HL1,N>|TC])
compress([HL1,HL2|TTL],[<HL1,1>|TC]) ←
                compress([HL2|TTL],TC)
```

Figure 4-4: A transformed version of *LP(compress-int-L)*

4.5 Transformation of a Logic Program

Correctness-preserving transformations of logic programs aim at optimizing the derived pure logic program into a time/space-efficient Prolog program. They are based on the procedural semantics (equivalence of SLDNF trees, the computation rule, the search rule), the available control predicates (*cut*, *fail*,…), and the implementation features (anonymous variables, tail recursion optimization, clause indexing, global parameters,…) of Prolog systems, as well as on partial evaluation, the properties of equality, and so on. These transformation rules are irrelevant in our context.

Example 4-13: For the *compress* predicate, Figure 4-4 gives a transformed version of the logic program in Figure 4-3.

5 Thesis

The objective of this thesis is (semi-)automatic logic algorithm synthesis. First, in Section 5.1, we define this objective in more detail and clearly state its boundaries. In order to motivate this research, we develop a series of sample problems in Section 5.2 and show that non-trivial issues have to be solved. This allows us to identify, in Section 5.3, the challenges of logic algorithm synthesis. Finally, in Section 5.4, we list our contributions towards attaining this objective, and outline the remainder of this thesis.

5.1 Objective

The objective of this thesis is (semi-)automatic logic algorithm synthesis. This means that our research is set in a logic programming framework, and that there is exclusive concern about the synthesis of correct logic algorithms, but no concern at all about algorithm efficiency, algorithm transformation, algorithm implementation, program transformation, or data structure reification. This modular approach achieves re-use of existing and forthcoming research on (logic) algorithm transformation and implementation. Indeed, the design of correct (logic) algorithms is already sufficiently hard, so there is no need to complicate matters with simultaneous considerations about time/space efficiency or implementability. The so-called "naive" version of the *reverse* algorithm is thus a perfectly convenient result, because (*i*) it is correct, (*ii*) it can be automatically transformed into a more efficient algorithm, and (*iii*) it can be automatically implemented into an executable form.

Moreover, the focus is on the synthesis of algorithms for the so-called algorithmic problems, and actually, to be even more precise, on the synthesis of recursive algorithms, where the intentions may correspond to a full-fledged relation (and not just to a total function). The following two assumptions are made. First, we assume that the specifier knows the intended relation, even if s/he doesn't have a formal definition of it. Second, we assume that the actually specified relation is a sub-set of the intended relation. This means that there is assumed to be no noise in specifications. Other aims are that the developed synthesis mechanism be able to synthesize entire families of algorithms from a single specification, and that a fair amount of algorithms knowledge and domain knowledge be available to it.

A first question that naturally comes to mind is: What specification language to use? As the relation entry in the specification language of [Deville 90] is non-formal, it is obvious that this language is only appropriate for manual construction of logic algorithms, but not for (semi-)automatic synthesis thereof. There are then basically three solutions: either augment the existing specification language with suitable entries, or develop a language that allows the writing of intermediate descriptions that are good starting points for automated synthesis, or develop a new specification language. In Chapter 6, we opt for the latter solution. The actually chosen language is irrelevant at this point, but the first two stages of the overall methodology now are:

- Stage A': Elaboration of a specification (in the new formalism);
- Stage B': Synthesis of a logic algorithm.

Once the new specification formalism developed, this thesis is mainly concerned with Stage B', and only marginally with Stage A'.

Compared to [Deville 90], we adopt the following two differences in attitude. First, we prefer to blur the distinction between the domain description and the relation definition of a specification. In the case of a complete specification, the specified relation is assumed to be identical to the intended relation. In the case of an incomplete specification, the specified relation is assumed to be a subset of the intended relation.

Second, at the logic algorithm level, we prefer to view domains as pre-conditions, as in [Deville 87]. This means that rather than making sure a logic algorithm yields *false* when its parameters are not compatible with the domain, one has to make sure the used predicates are called with parameters that are compatible with their domains. This approach is more natural, as it corresponds to standard practice at the logic program level.

Actually, considering that the introduction of explicit domain checking literals is often a minor task because most of the domain checking is already implicitly performed by the logic algorithm, and that it is often difficult otherwise, and that it is definitely not a truly creative part of algorithm design, and that the presence of these literals is a matter of personal taste anyway, we further restrict the scope of our objective to the synthesis of logic algorithms without domain checking.

5.2 Motivating Examples

In order to motivate this research, we pose a series of sample problems in Section 5.2.1, and give sample logic algorithms solving some of them in Section 5.2.2. Last, in Section 5.2.3, we give some comments on these logic algorithms.

5.2.1 Sample Problems

For the sake of conciseness, we adopt the following shorthand format for Deville's specification formalism:

<*procedure declaration*> iff <*relation definition*>,
where <*domain description*>.

Note that this format is restricted to the purpose of algorithm design, as no directionality information is given. Here follow the specifications of a series of sample problems that will be used throughout this thesis:

add(X,Y,S) iff *S* is the sum of *X* and *Y*,
where *X, Y, S* are integers.

append(A,B,C) iff *C* is the concatenation of *B* to the end of *A*,
where *A, B, C* are lists.

compress(L,C) iff *C* is a list of $<v_i,c_i>$ couples, such that the i^{th} plateau of *L* has c_i elements equal to v_i,
where *L* is a list, and *C* is a compact list.

delOddElems(L,R) iff *R* is *L* without its odd elements,
where *L, R* are integer lists.

efface(E,L,R) iff *R* is *L* without its first (existing) occurrence of *E*,
where *E* is a term, *L* is a non-empty list, and *R* is a list.

firstN(N,L,R) iff *R* is the first *N* elements of *L*,
where *N* is an integer, *L* is a list of at least *N* elements, and *R* is a list of *N* elements.

firstPlateau(L,P,S) iff *P* is the first plateau of *L*, and *S* is the corresponding suffix of *L*,
where *L* is a non-empty list, *P* is a plateau, and *S* is a list.

flatTree(T,L) iff *L* is the infix traversal of *T*,
where *L* is a list, and *B* is a binary tree.

insert(E,L,R) iff *R* is *L* with *E* inserted at the right place,
where *E* is an integer, *L* is an ascending integer list, and *R* is a non-empty ascending integer list.

length(L,N) iff *N* is the length of *L*,
where *N* is an integer, and *L* is a list.

member(E,L) iff *E* is an element of *L*,
where *E* is a term, and *L* is a list.

merge(A,B,C) iff *C* is the merger of *A* and *B*,
where *A, B, C* are ascending integer lists.

minimum(L,M) iff *M* is the minimum of *L*,
where *M* is an integer, and *L* is a non-empty integer list.

multiply(X,Y,P) iff *P* is the product of *X* and *Y*,
where *X, Y, P* are integers.

parity(L,R) iff *R* is *even* when *L* is of even length, and *odd* otherwise,
where *L* is a list, and *R* is a term.

partition(L,P,S,B) iff *S* (respectively *B*) contains the elements of *L* that are smaller than (respectively bigger than or equal to) the pivot *P*,
where *L, S, B* are integer lists, and *P* is an integer.

permutation(L,P) iff *P* is a permutation of *L*,
where *L, P* are lists.

plateau(N,E,P) iff *P* is a plateau of *N* elements equal to *E*,
where *N* is a positive integer, *E* a term, and *P* a non-empty list.

reverse(L,R) iff *R* is the reverse of *L*,
where *L, R* are lists.

sort(L,S) iff *S* is an ascendingly ordered permutation of *L*,
where *L, S* are integer lists.

split(L,F,S) iff *F* is the first half of *L*, and *S* is the second half of *L*,
where *L, F, S* are lists.

stuff(E,L,R) iff *R* is *L* with *E* inserted at a random place,
where *E* is a term, *L* is a list, and *R* is a non-empty list.

sum(L,S) iff *S* is the sum of the elements of *L*,
where *S* is an integer, and *L* is an integer list.

5.2.2 Sample Logic Algorithms

We now list sample logic algorithms in order to illustrate the variety of difficulties that arise. No domain checking literals have been added. The given logic algorithms may look somewhat contrived, and are definitely subject to many simplifying transformation opportunities, but these versions make subsequent analysis much easier. Also, these logic algorithms do not always exactly result from an application of the construction methodology of [Deville 90], as outlined in the previous chapter.

We first construct a few alternative logic algorithms for the *compress* predicate of Chapter 4. Then we list some logic algorithms for some of the problems posed in Section 5.2.1.

Alternative Logic Algorithms for the compress/2 Predicate

In order to give some more insights into Stage B of the existing methodology, we first construct a few alternative logic algorithms for the *compress* predicate.

Let's first reconsider Step 2 (Selection of a Well-Founded Relation), and follow the Extrinsic Heuristic when selecting a well-founded relation over the type of the induction param-

eter *L*. This means that we want to decompose *L* into something smaller in a way reflecting the structure of parameter *C*. Every element of *C* represents a "summary" of a plateau of *L*, so the idea is to decompose *L* by extracting its first plateau as head of *L*, and the corresponding suffix as tail of *L*. This decomposition is non-trivial, but considerably facilitates the rest of the construction. Step 3 is unaffected by this decision, and after Step 4, the result is:

```
compress(L,C) ⇔
    L=[]       ∧ C=[]
  ∨ L=[_|_]    ∧ firstPlateau(L,HL,TL)
               ∧ compress(TL,TC)
               ∧ plateau(N,E,HL) ∧ HC=<E,N>
               ∧ C=[HC|TC]
```

Logic Algorithm 5-1: *LA(compress-ext-L)*

Let's now reconsider Step 1 (Selection of an Induction Parameter), and select the other parameter, *C*, as induction parameter. At Step 2 (Selection of a Well-Founded Relation), we first follow the Intrinsic Heuristic, and select "is the tail of" as well-founded relation over the type of the induction parameter. Step 3 is straightforward, and after Step 4, the result is:

```
compress(L,C) ⇔
    C=[]       ∧ L=[]
  ∨ C=[_|_]    ∧ C=[HC|TC]
               ∧ compress(TL,TC)
               ∧ HC=<E,N> ∧ plateau(N,E,HL)
               ∧ firstPlateau(L,HL,TL)
```

Logic Algorithm 5-2: *LA(compress-int-C)*

Let's finally reconsider Step 2 (Selection of a Well-Founded Relation), and follow the Extrinsic Heuristic when selecting a well-founded relation over the type of induction parameter *C*. This means that we want to decompose *C* into something smaller in a way reflecting the structure of parameter *L*. Every element of *L* represents an increment by 1 of the counter in an element of *C*, so the idea is to decompose *C* by decrementing, if possible, the counter of its first element by 1. Step 3 is unaffected by this decision, and after Step 4, the result is:

```
compress(L,C) ⇔
    C=[]       ∧ L=[]
  ∨ C=[_|_]    ∧ decompose(C,HC,TC)
               ∧ compress(TL,TC)
               ∧ HL=HC
               ∧ L=[HL|TL]
decompose(C,HC,TC) ⇔
    C=[<E,s(N)>|T] ∧ N=0 ∧ HC=E ∧ TC=T
  ∨ C=[<E,s(N)>|T] ∧ N>0 ∧ HC=E ∧ TC=[<E,N>|T]
```

Logic Algorithm 5-3: *LA(compress-ext-C)*

Note that Logic Algorithm 5-1 and Logic Algorithm 5-2 are almost identical. Also, Logic Algorithm 4-3 and an unfolded version of Logic Algorithm 5-3 would be almost identical.

Logic Algorithms for Other Predicates

Here follow, in alphabetical order on the predicates, some logic algorithms for some problems posed in Section 5.2.1. We do not explain their design processes, but give some useful

comments on the results. The reader is invited to check that they are correct wrt their specifications. Unless otherwise noted, they have all been designed with the Intrinsic Heuristic.

Example 5-1: The following logic algorithm for *delOddElems(L,R)* has been designed with *L* as induction parameter. A design with *R* as induction parameter would yield a quite similar logic algorithm.

```
delOddElems(L,R) ⇔
    L=[]        ∧ R=[]
  ∨ L=[_|_]     ∧ L=[HL|TL]
                ∧ odd(HL)
                ∧ delOddElems(TL,TR)
                ∧ HR=_
                ∧ R=TR
  ∨ L=[_|_]     ∧ L=[HL|TL]
                ∧ ¬odd(HL)
                ∧ delOddElems(TL,TR)
                ∧ HR=HL
                ∧ R=[HR|TR]
```

Logic Algorithm 5-4: *LA(delOddElems-L)*

Example 5-2: The following logic algorithm for *efface(E,L,R)* has been designed with *L* as induction parameter. A design with *R* would yield a quite similar logic algorithm. Note that the non-minimal form gives rise to two cases, one of them without recursion. Also note that *E* could have been used instead of *TE* in the recursive atom, because both are unified with *HE*. Such a parameter is called an *auxiliary parameter*, because it has nothing to do with the inductive nature of the problem. Induction on an auxiliary parameter obviously is a bad idea.

```
efface(E,L,R) ⇔
    L=[_]       ∧ L=[HL] ∧ E=HL ∧ R=[]
  ∨ L=[_,_|_]   ∧ L=[HL|TL]
                ∧ HL=E
                ∧ E=HL ∧ R=TL
  ∨ L=[_,_|_]   ∧ L=[HL|TL]
                ∧ HL≠E
                ∧ efface(TE,TL,TR)
                ∧ HE=_ ∧ HR=HL
                ∧ E=TE ∧ R=[HR|TR]
```

Logic Algorithm 5-5: *LA(efface-L)*

Example 5-3: The following logic algorithm for *firstN(N,L,R)* has been designed with *L* as induction parameter. A design with *R* as induction parameter would yield a quite similar logic algorithm. The non-minimal form gives rise to two cases, one of them without a recursive atom. Note the relevant redundancy between the `N=0` atoms.

```
firstN(N,L,R) ⇔
    L=[]       ∧ N=0 ∧ R=[]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ N=0
               ∧ N=0 ∧ R=[]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ N>0
               ∧ firstN(TN,TL,TR)
               ∧ HN=_ ∧ HR=HL
               ∧ N=s(TN) ∧ R=[HR|TR]
```

Logic Algorithm 5-6: *LA(firstN-L)*

The following other logic algorithm for *firstN(N,L,R)* has been designed with *N* as induction parameter. Note the interesting variable sharing between *HL* and *HR*: their value is "invented", but common. Also note the relevant presence of a `true` atom.

```
firstN(N,L,R) ⇔
    N=0        ∧ L=_ ∧ R=[]
  ∨ N>0        ∧ N=s(TN) ∧ HN=N
               ∧ true
               ∧ firstN(TN,TL,TR)
               ∧ HL=A ∧ HR=A
               ∧ L=[HL|TL] ∧ R=[HR|TR]
```

Logic Algorithm 5-7: *LA(firstN-N)*

Example 5-4: The following logic algorithm for *insert(E,L,R)* has been designed with *L* as induction parameter. Parameter *E* is an auxiliary parameter. Note that the non-minimal form gives rise to two cases, one of them without a recursive atom.

```
insert(E,L,R) ⇔
    L=[]       ∧ E=_ ∧ R=[E]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ HL≥E
               ∧ E=_ ∧ R=[E|L]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ HL<E
               ∧ insert(TE,TL,TR)
               ∧ HE=_ ∧ HR=HL
               ∧ E=TE ∧ R=[HR|TR]
```

Logic Algorithm 5-8: *LA(insert-L)*

The following other logic algorithm for *insert(E,L,R)* has been designed with *R* as induction parameter. Note that the non-minimal form gives rise to two cases, one of them without a recursive atom.

```
insert(E,L,R) ⇔
    R=[_]      ∧ R=[A] ∧ E=A ∧ L=[]
  ∨ R=[_,_|_] ∧ R=[HR|TR]
              ∧ HR=E
              ∧ E=HR ∧ L=TR
  ∨ R=[_,_|_] ∧ R=[HR|TR]
              ∧ HR<E              {HR≠E would do as well}
              ∧ insert(TE,TL,TR)
              ∧ HE=_ ∧ HL=HR
              ∧ E=TE ∧ L=[HL|TL]
```

Logic Algorithm 5-9: *LA(insert-R)*

Example 5-5: The following logic algorithm for *member(E,L)* has been designed with *L* as induction parameter. Parameter *E* is an auxiliary parameter. Note that the non-minimal form gives rise to two cases, one of them without a recursive atom. Also note how the desired non-determinism is achieved.

```
member(E,L) ⇔
    L=[_]      ∧ L=[A] ∧ E=A
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ true
              ∧ E=HL
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ true
              ∧ member(TE,TL)
              ∧ HE=_
              ∧ E=TE
```

Logic Algorithm 5-10: *LA(member-L)*

Example 5-6: The following logic algorithm for *merge(A,B,C)* has been designed with the couple <*A,B*> as compound induction parameter. Note the different decomposition patterns. Parameter *A* or *B* alone as induction parameter does not lead to a successful design. This is however possible with *C* alone, but we do not list that logic algorithm here.

```
merge(A,B,C) ⇔
    A=[]       ∧ C=B
  ∨ B=[]       ∧ C=A
  ∨ A=[_|_] ∧ B=[_|_]
               ∧ A=[HA|TA] ∧ B=[HB|TB]
               ∧ HA=HB
               ∧ merge(TA,TB,TC)
               ∧ HC=HA ∧ HC=HB
               ∧ C=[HC,HC|TC]
  ∨ A=[_|_] ∧ B=[_|_]
               ∧ A=[HA|TA] ∧ B=TB
               ∧ HA<HB ∧ B=[HB|_]
               ∧ merge(TA,TB,TC)
               ∧ HC=HA
               ∧ C=[HC|TC]
  ∨ A=[_|_] ∧ B=[_|_]
               ∧ A=TA ∧ B=[HB|TB]
               ∧ HA>HB ∧ A=[HA|_]
               ∧ merge(TA,TB,TC)
               ∧ HC=HB
               ∧ C=[HC|TC]
```

Logic Algorithm 5-11: *LA(merge-<A,B>)*

Example 5-7: The following logic algorithm for *parity(L,R)* has been designed with *L* as induction parameter. Note that there are two minimal forms, and that the non-minimal form is decomposed by taking two elements away from *L*.

```
parity(L,R) ⇔
    L=[]       ∧ R=even
  ∨ L=[_]      ∧ R=odd
  ∨ L=[_,_|_] ∧ L=[HL1,HL2|TL]
               ∧ true
               ∧ parity(TL,TR)
               ∧ HR=_
               ∧ R=TR
```

Logic Algorithm 5-12: *LA(parity-L)*

Example 5-8: The following logic algorithm for *partition(L,P,S,B)* has been designed with *L* as induction parameter. Parameter *P* is an auxiliary parameter.

```
partition(L,P,S,B) ⇔
    L=[]       ∧ P=_ ∧ S=[] ∧ B=[]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ HL≥P
               ∧ partition(TL,TP,TS,TB)
               ∧ HP=_ ∧ HS=_ ∧ HB=HL
               ∧ P=TP ∧ S=TS ∧ B=[HB|TB]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ HL<P
               ∧ partition(TL,TP,TS,TB)
               ∧ HP=_ ∧ HS=HL ∧ HB=_
               ∧ P=TP ∧ S=[HS|TS] ∧ B=TB
```

Logic Algorithm 5-13: *LA(partition-L)*

Example 5-9: The following logic algorithm for *permutation(L,P)* has been designed with *L* as induction parameter. Because of the symmetry of the underlying relation, a design with *P* as induction parameter would yield the same logic algorithm. Note that the use of *efface(HP,P,TP)* rather than *stuff(HP,TP,P)* avoids redundant solutions at the logic program level.

```
permutation(L,P) ⇔
    L=[]       ∧ P=[]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ permutation(TL,TP)
               ∧ HP=HL
               ∧ efface(HP,P,TP)
```

Logic Algorithm 5-14: *LA(permutation-L)*

Example 5-10: The following logic algorithm for *plateau(N,E,P)* has been designed with *N* as induction parameter. A design with *P* as induction parameter would yield a quite similar logic algorithm. Note the intricate way in which *HP* is unified with the correct value. Parameter *E* is an auxiliary parameter.

```
plateau(N,E,P) ⇔
    N=1        ∧ E=_ ∧ P=[E]
  ∨ N>1        ∧ N=s(TN) ∧ HN=N
               ∧ plateau(TN,TE,TP)
               ∧ HE=A ∧ HP=A
               ∧ E=TE ∧ E=HE ∧ P=[HP|TP]
```

Logic Algorithm 5-15: *LA(plateau-N)*

Example 5-11: The following logic algorithm for *sort(L,S)* has been designed with *L* as induction parameter, following the Intrinsic Heuristic. The result is the Insertion-Sort algorithm.

```
sort(L,S) ⇔
    L=[]       ∧ S=[]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ sort(TL,TS)
               ∧ HS=HL
               ∧ insert(HS,TS,S)
```

Logic Algorithm 5-16: *LA(sort-int-L)* {Insertion-Sort}

The following other logic algorithm for *sort(L,S)* has been designed with *L* as induction parameter, following the Extrinsic Heuristic by partitioning *L* according to some pivot element. The result is the Quick-Sort algorithm.

```
sort(L,S) ⇔
    L=[]       ∧ S=[]
  ∨ L=[_|_]    ∧ L=[HL|T] ∧ partition(T,HL,TL1,TL2)
               ∧ sort(TL1,TS1) ∧ sort(TL2,TS2)
               ∧ HS=HL
               ∧ append(TS1,[HS|TS2],S)
```

Logic Algorithm 5-17: *LA(sort-ext-L)* {Quick-Sort}

The following other logic algorithm for *sort(L,S)* has been designed with *L* as induction parameter, following the Extrinsic Heuristic by splitting *L* into two halves. The result is the Merge-Sort algorithm.

```
sort(L,S) ⇔
    L=[]       ∧ S=[]
  ∨ L=[_|_]    ∧ split(L,TL1,TL2)
               ∧ sort(TL1,TS1) ∧ sort(TL2,TS2)
               ∧ merge(TS1,TS2,S)
```

Logic Algorithm 5-18: *LA(sort-ext-L)* {Merge-Sort}

Example 5-12: The following logic algorithm for *split(L,F,S)* has been designed by a totally different method, namely introduction of an additional parameter, plus logic algorithm design by structural induction for the new problem.

```
split(L,F,S) ⇔
    split(L,L,F,S)

split(L,M,F,S) ⇔
    M=[]       ∧ F=[] ∧ S=L
  ∨ M=[_]      ∧ F=[] ∧ S=L
  ∨ M=[_,_|_]  ∧ M=[HM1,HM2|TM] ∧ L=[HL|TL]
               ∧ true
               ∧ split(TL,TM,TF,TS)
               ∧ HF=HL ∧ HS=_
               ∧ F=[HF|TF] ∧ S=TS
```

Logic Algorithm 5-19: *LA(split)*

5.2.3 Some Comments on Logic Algorithms

Several aspects of logic algorithm design can be discussed now. We first propose a useful terminology for logic algorithm classification, and then show in what sense minimal cases and non-recursive non-minimal cases, though syntactically similar, are totally different concepts.

Logic Algorithm Classification

Logic Algorithms can be classified along five dimensions, namely according to how the induction parameter is decomposed, according to how much of the induction parameter is actually traversed, according to how many times the induction parameter is traversed, according to the determinism of the defined relation given a ground value of the induction parameter, and according to the kind of manipulations performed on the constituents (constants and variables) of the parameters.

Regarding the decomposition of the induction parameter, we distinguish between the following three categories of logic algorithms:

- *intrinsic-decomposition* logic algorithms: if X is decomposed into $h \geq 1$ heads and $t \geq 1$ tails in a manner reflecting the definition of the type of X; for instance, decompose a list L into its head HL and tail TL;
- *extrinsic-decomposition* logic algorithms: if X is decomposed into $h \geq 0$ heads and $t \geq 1$ tails in a manner reflecting the definition of the type of some other parameter than X, or reflecting the intended relation; for instance, partition an integer-list L according to a pivot P into a list S of elements smaller than P and a list B of elements bigger than or equal to P;
- *logarithmic-decomposition* logic algorithms: if X is decomposed into $h = 0$ heads and $t \geq 2$ tails of about equal size; for instance, split a list L into two halves A and B.

An intrinsic decomposition reflects a well-founded relation selected via the Intrinsic Heuristic, and an extrinsic or logarithmic decomposition reflects a well-founded relation selected via the Extrinsic Heuristic (see Chapter 4). Sample classifications are given below.

Regarding the scope of the traversal of the induction parameter, we here distinguish between the following two categories of logic algorithms:

- *complete-traversal* logic algorithms: all elements of the induction parameter are visited;
- *prefix-traversal* logic algorithms: only the first few elements of the induction parameter are visited.

Sample classifications are given below.

Regarding the number of times the induction parameter is actually traversed, we distinguish between the following two categories of logic algorithms:

- *single-loop* logic algorithms: only one traversal of the induction parameter is being performed at any moment;
- *multiple-loop* logic algorithms: at least two nested traversals of the induction parameter are performed at some moment.

Sample classifications are given below.

Regarding the determinism of a logic algorithm given ground values of the induction parameter and the auxiliary parameter(s), we define the following notions.

Definition 5-1: A *scalar* is either an integer, or the special symbol $*$ (representing any finite integer), or the special symbol ∞ (representing infinity).

Definition 5-2: Let *min* and *max* be two scalars. Let *LP(r)* be the logic program derived from a logic algorithm *LA(r)*. Let ***r(t)*** be an atom where the induction parameter and the auxiliary parameter(s) are ground, and all other parameters are variables. We say that *LA(r)* is *min – max deterministic* iff there are between *min* and *max* ground instances of ***r(t)*** for which there exists an SLDNF refutation using *LP(r)*.

The following terminology is then useful for classifying logic algorithms according to their determinism:

- *deterministic* logic algorithms are 0 – 1 deterministic;
- *fully-deterministic* logic algorithms are 1 – 1 deterministic;
- *non-deterministic* logic algorithms are *min – max* deterministic, where *min* is an integer greater than 0, and *max* is either an integer greater than *min*, or $*$, or ∞;
- *finite* logic algorithms are *min – max* deterministic, where *min* is an integer, and *max* is either an integer greater than *min*, or $*$;
- *infinite* logic algorithms are *min* – ∞ deterministic, where *min* is any scalar.

Sample classifications are given below.

Finally, regarding the kind of manipulations that are performed on the constituents of the parameters, we distinguish between the following two categories of logic algorithms:

- *structural-manipulation* logic algorithms: some—if not all—constituents of the induction parameter are shuffled around unconditionally, because their values are irrelevant, for the construction of the other parameters;
- *semantic-manipulation* logic algorithms: some—if not all—constituents of the induction parameter are shuffled around conditionally, because their values are relevant, or even new constituents are created, for the construction of the other parameters.

This last dimension is actually also applicable for classifying relations, as no design choice affects into which category the resulting logic algorithms fall.

Example 5-13: Table 5-1 charts the features of the relations for which logic algorithms have been given so far. The columns list (from left to right):

- the arity of the relation;
- the presence of auxiliary parameters (denoted by their names), or the absence (denoted *no*) of auxiliary parameters;
- the kind of the manipulations on the parameters. ♦

But this ambivalence does not hold for the first four dimensions, because a relation might give rise to several logic algorithms that are classified differently along these dimensions.

Example 5-14: Table 5-2 charts the features of the logic algorithms given so far. The columns list (from left to right):

- the name of the selected induction parameter;
- the kind of the selected induction parameter (where *s* stands for simple induction parameter, and *c* stands for compound induction parameter);
- the kind of traversal performed on the induction parameter (where *comp* stands for complete-traversal, and *pre* stands for prefix-traversal);
- the determinism of the logic algorithm given ground values of the induction parameter and the auxiliary parameter(s);
- the number of structural forms of the induction parameter, represented as $a+b$, where a is the number of minimal forms, and b is the number of non-minimal forms;
- the selected strategy of decomposition of the induction parameter (where *int* stands for intrinsic decomposition, *ext* stands for extrinsic decomposition, and *log* stands for logarithmic decomposition);

Table 5-1: Summary of the features of some sample relations

	Arity	AP	Manipulation
compress	2	no	semantic
delOddElems	2	no	semantic
efface	3	*E*	semantic
firstN	3	no	structural
firstPlateau	3	no	semantic
insert	3	*E*	semantic
member	2	*E*	structural
merge	3	no	semantic
parity	2	*R*	structural
partition	4	*P*	semantic
permutation	2	no	structural
plateau	3	*E*	semantic
sort	2	no	semantic
split	4	no	structural

- the number of cases of the logic algorithm, represented as $u+v+w$, where u is the number of minimal cases, v is the number of non-minimal non-recursive cases, and w is the number of non-minimal recursive cases;
- the number of loops performed by the logic algorithm (where 1 indicates that it is a single-loop logic algorithm, and integers greater than 1 indicates that it is a multiple-loop logic algorithm);
- the number of the logic algorithm within this thesis, and the page where it can be found.

The first four features are independent of the selected decomposition strategy. The numbers of cases and loops depend on the selected decomposition strategy. ♦

Minimal Cases and Non-Recursive, Non-Minimal Cases

As the logic algorithms of this thesis exhibit, not all non-minimal cases are recursive. Indeed, prefix-scan logic algorithms have such cases. But some structural cases of logic algorithms in Chapter 4, or [Deville 90], seem hard to classify. There are several potential reasons to this.

A first reason is that non-recursive, non-minimal cases of prefix-scan logic algorithms can often be merged with their minimal cases. The resulting logic algorithm looks like it has no minimal case. It exhibits cases whose structural forms are not mutually exclusive, and is thus easy to detect as being a rewriting from the "canonical" version.

Example 5-15: Logic Algorithm 5-5 could be rewritten as follows:

Table 5-2: Summary of the features of some sample logic algorithms

	IP	K	Trav	Det	Fms	Dec	Cases	Lps	LA #, page #
compress	*L*	s	comp	1 – 1	1+1	int	1+0+2	1	4-3, page 61
						ext	1+0+1	2	5-1, page 68
	C	s	comp	1 – 1	1+1	int	1+0+1	2	5-2, page 68
						ext	1+0+1	1	5-3, page 68
delOddElems	*L*	s	comp	1 – 1	1+1	int	1+0+2	1	5-4, page 69
efface	*L*	s	pre	1 – 1	1+1	int	1+1+1	1	5-5, page 69
firstN	*L*	s	pre	1 – ∗	1+1	int	1+1+1	1	5-6, page 70
	N	s	comp	1 – 1	1+1	int	1+0+1	1	5-7, page 70
firstPlateau	*L*	s	pre	1 – 1	1+1	int	1+1+1	1	4-2, page 58
insert	*L*	s	pre	1 – 1	1+1	int	1+1+1	1	5-8, page 70
	R	s	pre	1 – 1	1+1	int	1+1+1	1	5-9, page 71
member	*L*	s	comp	1 – ∗	1+1	int	1+1+1	1	5-10, page 71
merge	*A*, *B*	c	comp	1 – 1	2+1	int	2+0+3	1	5-11, page 72
parity	*L*	s	comp	1 – 1	2+1	int	2+0+1	1	5-12, page 72
partition	*L*	s	comp	1 – 1	1+1	int	1+0+2	1	5-13, page 73
permutation	*L*	s	comp	1 – ∗	1+1	int	1+0+1	2	5-14, page 73
plateau	*N*	s	comp	1 – 1	1+1	int	1+0+1	1	5-15, page 73
sort	*L*	s	comp	1 – 1	1+1	int	1+0+1	2	5-16, page 74
						ext	1+0+1	2	5-17, page 74
						log	1+0+1	2	5-18, page 74
split	*M*	s	comp	1 – 1	2+1	int	2+0+1	1	5-19, page 74

```
efface(E,L,R) ⇔
    L=[_|_]    ∧ L=[HL|TL]
               ∧ HL=E
               ∧ E=HL ∧ R=TL
  ∨ L=[_,_|_] ∧ L=[HL|TL]
               ∧ HL≠E
               ∧ efface(TE,TL,TR)
               ∧ HE=_ ∧ HR=HL
               ∧ E=TE ∧ E=HE ∧ R=[HR|TR]
```

The form *L*=[_|_] is not a minimal form,as it overlaps with the other form. It rather results from a merger of the minimal case and the non-recursive, non-minimal case. ♦

Another reason for non-recursive, non-minimal cases is recursion elimination by partial evaluation. The result is a non-minimal case that looks like a minimal case. This is harder to detect, since the cases still exhibit mutually exclusive structural forms. There is of course no limit to creating non-recursive, non-minimal cases by partial evaluation.

Example 5-16: Logic Algorithm 4-1 is actually a rewriting of:

```
compress(L,C) ⇔
    L=[]        ∧ C=[]
  ∨ L=[_|_]     ∧ L=[HL|TL]
                ∧ ( TL=[] ) ∨ ( L=[HL1,HL2|_] ∧ HL1≠HL2 )
                ∧ compress(TL,TC)
                ∧ HC=<HL,1>
                ∧ C=[HC|TC]
  ∨ L=[_|_]     ∧ L=[HL|TL]
                ∧ L=[HL1,HL2|_] ∧ HL1=HL2
                ∧ compress(TL,TC)
                ∧ HC=_
                ∧ C=[<V,s(N)>|TTC] ∧ TC=[<V,N>|TTC]
```

Indeed, we have that *TL*=[] iff *TC*=[], so the rewriting was correct. But it was also misleading, as the fact that there really is only one minimal case and only one non-minimal form was not apparent at all. ♦

It is important to understand that such logic algorithms with non-recursive, non-minimal cases are the result of re-writing "canonical" logic algorithms, rather than unpleasant aberrations.

5.3 Challenges

Looking at the sheer variety of possible logic algorithms, some of them in the preceding section, the automatic synthesis of logic algorithms looks like quite a formidable task. Indeed, the following difficulties need to be tackled:

- *What induction parameter to select? How to discover compound induction parameters? According to what well-founded relation to decompose the induction parameter?* Ideally, a synthesis mechanism should be able to design the whole family of possible logic algorithms for a given problem. For instance, given the sorting problem, at least the three logic algorithms above should be designed.
- *How many structural forms are there?* The number of minimal and non-minimal structural forms is specific to each logic algorithm. While most logic algorithms seem to have one minimal and one non-minimal form, this does not always hold, as illustrated by the logic algorithms for *parity*/3 and *split*/4.
- *What are the structural forms?* The type of the induction parameter is not sufficient to infer its structural forms: these are actually dependent on the domain of the induction parameter, and are thus a problem-specific issue.
- *Into how many cases is each structural case divided? How to discriminate between these sub-cases?* Many of the logic algorithms listed above fork their non-minimal case into two sub-cases.
- *How to detect that recursion is useless in some non-minimal sub-cases?* Sometimes, the desired result is obtained before reducing the induction parameter to a minimal form, and no recursion is then needed: this happens for instance in the logic algorithms for *efface*/3 and *insert*/3. The existence or not of useless recursion is dependent on the selected induction parameter, as illustrated by the two logic algorithms for *firstN*/3.

- *How to "invent" or re-use appropriate predicates? How to implement "invented" predicates?* The combination of partial results often is a full-scale problem by itself, as illustrated by the logic algorithms for *compress*/2 (induction on *C*), *permutation*/2, and *sort*/2. The same holds for a decomposition following the Extrinsic Heuristic. This is known as the predicate invention problem.
- *How to discover which parameters are auxiliary parameters?* Problems such as *efface*/3, *insert*/3, *member*/2, *partition*/4, and *plateau*/3 have auxiliary parameters: unless this is stated somewhere, considerable design effort may go into detecting this. The logic algorithms listed above for these problems actually are versions for which the detection was not yet done.
- *How to synthesize logic algorithms that are non-deterministic*, such as those for *member*/2 and *permutation*/2? As seen in Section 2.4, this is known to be a hard problem for deductive synthesis approaches.
- *How to achieve a synthesis that yields logic algorithms that are correct wrt their specifications?* This is another tough problem.

and so on. This list of challenges is impressive. The answers depend of course a lot on the chosen specification language. Once that language defined, and a synthesis mechanism developed for it, we evaluate the results to see to what extent we have brought answers to these challenges.

5.4 Results and Contributions

Our synthesis mechanism builds upon a wide variety of ideas found in algorithm design, inductive inference, deductive inference, theorem proving, and so on.

In Part II, we provide the *building blocks* that are used later (in Part III) for the development of a logic algorithm synthesis mechanism. These building blocks are a language for expressing incomplete specifications (Chapter 6), a complete theoretical framework for the formulation of stepwise synthesis strategies (Chapter 7), an introduction to the notion of algorithm schemas and their usage in algorithm synthesis (Chapter 8), a method for deductively synthesizing parts of logic algorithms (Chapter 9), and another method for inductively synthesizing parts of logic algorithms (Chapter 10). Note that Chapters 7 to 10 may be read in any order. More specifically:

In Chapter 6, we define a *specification approach* that is based on the notions of *examples and properties*. It requires multiple, ground, single-predicate, relational, positive and negative, pre-synthesis examples that are chosen in a consistent way by a human specifier who knows the intended relation. The presence of properties (whose actual language is application-specific, and thus left unspecified for a while) is meant to overcome the problems of ambiguity and limited expressive power of examples, while still preserving their virtues of naturalness and conciseness. Such specification languages are quite expressive and readable. Specifications by examples and properties are usually incomplete, and hence ambiguous, but minimal. There is a danger of internal inconsistency and redundancy in such specifications, though. This specification approach holds the promise of faster and more dependable synthesis than from specifications by examples alone.

In Chapter 7, we develop a complete *theoretical framework for stepwise synthesis of logic algorithms from specifications by examples and properties*. Three layers of new correctness criteria relating intentions, specifications, and logic algorithms are introduced. Comparison criteria relating logic algorithms in terms of semantic or syntactic generality are then proposed. All these criteria provide an adequate structure for the formulation of stepwise synthesis strategies, be they incremental (examples and properties are presented one-by-one) or

non-incremental (examples and properties are presented all-at-once). A particular non-incremental strategy is developed in greater detail for use in the sequel.

In Chapter 8, we discuss *algorithm schemas* as an important support for algorithm design. Such schemas are an old idea in computer science, but one of the major ideas of this thesis is that schema-independent methods can be developed for the synthesis of instances of the variables of schemas. Such methods may be merely based on databases of useful instances, but more sophisticated methods would perform actual computations for inferring such instances. Such computations would be based on the specifications and the algorithm designed so far. Several methods of such a tool-box might be applicable at each step, thus yielding opportunities for user interaction, or for the application of heuristics. We thus advocate a very disciplined approach to algorithm synthesis: rather than using a uniform method for instantiating all variables of a given schema (possibly without any awareness of such a schema), one should deploy for each variable the best-suited method. In other words, we propose to view research on automatic programming as:

(1) the search for adequate schemas;
(2) the development of useful methods of predicate variable instantiation;
(3) the discovery of interesting mappings between these methods and the variables of these schemas.

As many methods would be schema-independent, and hence variable-independent, one could even investigate synthesis methodologies that are parameterized on schemas. In other words, a first step would be to select an appropriate schema, and the subsequent steps would be either a hardwired sequence (specific to the selected schema) of applications of methods, or a user-guided selection of variables and methods. Our grand view of algorithm synthesis systems thus is one of a large workbench with a disparate set of specialized methods and a set of schemas that covers (as much as possible of) the space of all possible algorithms.

In Chapter 9, we develop the *Proofs-as-Programs Method*, which deductively adds literals to a logic algorithm so that it satisfies a given set of properties. The added literals are extracted from the proof that the given logic algorithm is complete wrt these properties. This is not a new problem, but our method extends some existing methods, and is completely different from others. This method is part of our tool-box for instantiating variables of an algorithm schema.

In Chapter 10, we develop the *Most-Specific-Generalization Method*, which inductively synthesizes a logic algorithm from a set of examples. The intended relation, though unknown as a whole, is however known to feature a given data-flow pattern between its parameters. This method is applicable if the intended relation can be expressed by a logic algorithm that is defined solely in terms of the =/2 primitive. The synthesized logic algorithm is correct wrt a "natural extension" of the given examples. Note that this method is also part of our tool-box for instantiating variables of an algorithm schema, but not meant to be a solution to the more general problem of synthesis from specifications by examples. Indeed, our problem statement is much more specific here, which justifies the highly specialized, new method.

Most of these building blocks are thus not necessarily new (to algorithm synthesis), but they are here combined in a novel way, if not extended, and associated with some new results.

In Part III, we develop an actual *logic algorithm synthesis mechanism* from incomplete specifications by examples and properties (as seen in Chapter 6). It fits the particular non-incremental synthesis strategy (presented in Chapter 7), is guided by a divide-and-conquer algorithm schema (as seen in Chapter 8), and uses the tool-box of methods (developed in Chapter 9 and Chapter 10). More specifically:

In Chapter 11, we motivate the desired *features* of the mechanism (such as the actual language for the properties), and argue for a series of preliminary *restrictions*, so as to keep the

presentation simple until the discussion of its extensions. We also give an intuitive *overview* of the entire synthesis mechanism by illustrating it on a sample execution, so as to give the reader the feel for its working.

In Chapter 12, we give full detail about the *expansion phase of synthesis*, that is the first four steps of the mechanism. These steps are rather straightforward, and do not require any sophisticated methods.

In Chapter 13, we give full detail about the *reduction phase of synthesis*, that is the remaining three steps of the mechanism. These steps are the truly creative ones, and require the sophisticated methods of the tool-box developed in Chapter 9 and Chapter 10.

In Chapter 14, we provide a detailed *evaluation* of the obtained synthesis mechanism wrt the challenges of the previous section. Domain generality and full automation are sacrificed for the sake of better end-user orientation, and because this is a most reasonable approach with incomplete specifications anyway. Incorporated knowledge sources are the algorithms knowledge of the divide-and-conquer schema, and domain knowledge about types. Whole algorithm families can be synthesized from a single specification, because some synthesis steps are non-deterministic. The predicate invention problem is tackled at various points during the synthesis, namely by re-use of predefined predicates, by extraction from the properties, and by inference of specifications for sub-problems plus subsequent synthesis of sub-algorithms therefrom. The kinds of inference used during synthesis are inductive inference and deductive inference. The synthesis mechanism is a hybrid of transformational synthesis (due to its stepwise approach), proofs-as-programs synthesis, knowledge-based synthesis, bottom-up approximation-driven empirical learning, and Summers' recurrence detection mechanism. The synthesis of multiple-loop logic algorithms and of non-deterministic logic algorithms is possible. We also discuss some *extensions* to the synthesis mechanism (such as the handling of negation, of auxiliary parameters, and of other schemas). Finally, we outline a *methodology for choosing "good" examples and properties*, which, when followed, increases dependability and speed of synthesis, and decreases the need for interaction with the specifier. The mechanism seems very robust to example ordering and example choice, though.

A prototype implementation of this synthesis mechanism is being developed. It is called SYNAPSE (*SYNthesis of logic Algorithms from PropertieS and Examples*), and is written in portable Prolog.

II BUILDING BLOCKS

In this second part, we provide the building blocks for a solution to our objective. Thus, in Chapter 6, we define a generic specification approach that is based on examples and properties. Next, in Chapter 7, we develop a theoretical framework for stepwise synthesis of logic algorithms from such specifications, and address the aspects of correctness and comparison of logic algorithms. In Chapter 8, we introduce logic algorithm schemata as an important support for logic algorithm design. Chapter 9 is about the Proofs-as-Programs Method, which deductively enhances a logic algorithm such that it satisfies certain correctness criteria wrt a given set of properties. Finally, Chapter 10 is about the Most-Specific-Generalization Method, which inductively infers a logic algorithm from a given set of examples. Chapters 7 to 10 may be read in any order.

6 A Specification Approach

The first building block is the actual specification formalism used to elaborate the input to algorithm synthesis. An arbitrary choice has been made here to investigate synthesis from incomplete specifications. Starting from the pros and cons of specifications by examples, and of specifications by axioms, as outlined in Part I, we define, in Section 6.1, a specification approach that is based on examples and properties. Then, in Section 6.2, we illustrate this approach on a few sample problems. Future work and related work are discussed in Section 6.3 and Section 6.4, respectively, before drawing some conclusions in Section 6.5.

6.1 Specifications by Examples and Properties

When contrasting the pros and cons of specifications by axioms (as seen in Section 2.1) and specifications by examples (as seen in Section 3.1), it turns out that these specification approaches are quite complementary, each alleviating the drawbacks of the other by its own advantages. This gives rise to the idea of combining these two specification approaches so as to preserve only their benefits, while diminishing their disadvantages. But we must bear in mind the fundamental difference between the two approaches, namely specification completeness and specification incompleteness.

Merely amalgamating the two formalisms into specifications by axioms and examples is thus not a good idea, as the incomplete example-set only plays an illustrative side-role to the complete axiom-set.

But, on the other hand, one could aim at incomplete specifications and add some weaker form of axioms to the examples so as to overcome the weaknesses of specifications by examples only. We call *properties* such a relaxed form of axioms, and only require them to be written in (some subset of) logic. Indeed, until Part III, we do not restrict ourselves to any syntax or required computational content of properties. We only assume that properties are an incomplete source of information. Actually, in case properties were a complete source of information, most of the results hereafter would remain valid, but not always be relevant.

Let $\mathcal{R}$ be the relation one has in mind when elaborating a specification of a procedure for predicate r. We call $\mathcal{R}$ the *intended relation*, in contrast to the relation actually specified, called the *specified relation*. This distinction is very important in general, but crucial with incomplete specifications, where one deliberately admits a gap between the two.

The following three assumptions are crucial in the sequel. First, we only aim at specifying relations that give rise to recursive algorithms. This excludes many of the concept descriptions aimed at by the machine learning community. Second, we assume the specifier knows $\mathcal{R}$, even if s/he doesn't have a formal definition of it. This also precludes some scenarios envisaged by the machine learning community. Third, we assume that the specified relation is a sub-set of the intended relation. This means that there is assumed to be no noise in specifications. These assumptions define a highly specialized niche within the learning problem, and allow hence some specific choices in the sequel.

Definition 6-1: A *specification by examples and properties* of a procedure for predicate r/n, denoted $EP(r)$, consists of:

- a set $\mathcal{E}(r)$ of ground examples of r/n, partitioned into:
  - a set $\mathcal{E}^+(r)$ of positive examples of r/n (that is, ground atoms whose n-tuples are supposed to belong to $\mathcal{R}$);
  - a set $\mathcal{E}^-(r)$ of negative examples of r/n (that is, negated ground atoms whose n-tuples are supposed not to belong to $\mathcal{R}$);
- a set $\mathcal{P}(r)$ of properties (first-order logic statements) of r/n.

The (unique) specified relation is defined as follows:

Definition 6-2: The *specified relation* of a specification by examples and properties is the set of tuples extracted from the set of its Herbrand-logical consequences.

It is interesting to note that, very often, examples are properties, because the example language is a subset of the property language. In these cases, one could then simply talk about specifications by properties. But a specification language doesn't prejudice on how the specification information is actually used in algorithm design: indeed, an algorithm designer might use examples in a way that is totally different from the way non-example properties are used. This is why, in case the language subset relationship holds, we still prefer to talk about specifications by examples and properties. The assumption then is that the word "property" refers to properties that are not examples.

6.2 Sample Specifications by Examples and Properties

Let's illustrate the notion of specifications by examples and properties by a few sample specifications. The chosen language for properties is, for the sake of illustration, Horn clauses that are non-recursive, that have a head with the predicate of the examples, and that may have negated atoms in their bodies. Universal quantifiers are usually dropped for convenience. Also, negative examples are not required. Note that most of Part II is independent of these choices. Our claim is that such properties and examples, if carefully chosen, embody the minimal knowledge that doesn't give away the solution, but is sufficient to successful algorithm design.

In the first two sample specifications hereafter, we also illustrate how properties can be elaborated as generalized versions of examples.

Example 6-1: A sample version of *EP(compress)* is:

$\mathcal{E}$(compress) = { compress([],[]) (E_1)
compress([a],[a,1]) (E_2)
compress([b,b],[b,2]) (E_3)
compress([c,d],[c,1,d,1]) (E_4)
compress([e,e,e],[e,3]) (E_5)
compress([f,f,g],[f,2,g,1]) (E_6)
compress([h,i,i],[h,1,i,2]) (E_7)
compress([j,k,m],[j,1,k,1,m,1]) } (E_8)

$\mathcal{P}$(compress) = { compress([X],[X,1]) (P_1)
compress([X,Y],[X,2]) ⇐ X=Y (P_2)
compress([X,Y],[X,1,Y,1]) ⇐ X≠Y } (P_3)

Note that, for ease of syntax, we have changed the format of compact lists to lists whose elements alternately are values and counters.

How to elaborate these properties?

Note that properties P_1 to P_3 generalize examples E_2 to E_4, respectively. Example E_1 can't be generalized. So what about generalizing examples E_5 to E_8? The Horn clause:

$$compress([X, Y, Z], [X, 2, Z, 1]) \Leftarrow X{=}Y \wedge Y{\neq}Z$$

generalizes example E_6, and is a legal property. But it doesn't introduce any predicates the three properties above don't already introduce: it could thus be considered a superfluous property. Similarly, properties generalizing examples E_5, E_7, or E_8 wouldn't provide any new information.

But one could also think about writing properties that generalize more than the given examples, say with the first parameter being a list of any length. Rather than only generalizing the problem-specific constants *a, b,…* of the examples into variables *X, Y,…*, this consists in also generalizing the predefined constant *nil* into a variable, say *T*. The Horn clauses:

$$compress([X \mid T], [X, 1 \mid U]) \Leftarrow T{\neq}[X \mid \_] \wedge compress(T, U)$$
$$compress([X \mid T], [X, s(N) \mid U]) \Leftarrow compress(T, [X, N \mid U])$$
$$compress([X, Y \mid T], [X, 2 \mid U]) \Leftarrow X{=}Y \wedge T{\neq}[Y \mid \_] \wedge compress(T, U)$$
$$compress([X, Y \mid T], [X, 1 \mid U]) \Leftarrow X{\neq}Y \wedge compress([Y \mid T], U)$$
$$compress([X, Y \mid T], [X, 1, Y, 1 \mid U]) \Leftarrow X{\neq}Y \wedge T{\neq}[Y \mid \_] \wedge compress(T, U)$$

state constraints on compressing lists of at least one, or at least two, elements. But, for us, they are not properties because of the non-recursion restriction. Indeed, these statements are already almost identical to clauses of a logic program that correctly implements the given problem. Moreover, the elaboration of correct recursive properties is almost as difficult as writing the logic algorithm itself. This is clearly not what one would expect from an incomplete specification, and thus speaks in favor of disallowing recursive properties. If one is given such statements where the gap to a correct logic program is quite small, then the techniques advocated in this thesis are not adequate, and a purely deductive approach would be more reasonable.

Evaluation of this specification

When studying this first sample specification, it becomes easily apparent how it improves its examples-only counterpart: properties allow the specifier to make explicit what s/he perfectly knows, but can't express by examples alone. Especially the last two properties in the given property set embody such additional knowledge: it is equality or disequality of the first two elements of the non-compressed list that discriminate between having the first counter of the compressed list being 1 or larger than 1. This knowledge has otherwise to be guessed by the algorithm designer, which is dangerous (risk of wrong guesses) and time-consuming (enumeration of all possible guesses). The addition of disambiguating properties thus holds the promise of faster and more dependable algorithm design. ♦

Example 6-2: A sample version of *EP(firstPlateau)* is:

```
ℰ(firstPlateau) = { firstPlateau([a],[a],[])                (E1)
                    firstPlateau([b,b],[b,b],[])            (E2)
                    firstPlateau([c,d],[c],[d])             (E3)
                    firstPlateau([e,f,g],[e],[f,g])         (E4)
                    firstPlateau([h,i,i],[h],[i,i])         (E5)
                    firstPlateau([j,j,k],[j,j],[k])         (E6)
                    firstPlateau([m,m,m],[m,m,m],[]) }      (E7)
𝒫(firstPlateau) = { firstPlateau([X],[X],[])                (P1)
                    firstPlateau([X,Y],[X,Y],[]) ⇐ X=Y      (P2)
                    firstPlateau([X,Y],[X], [Y]) ⇐ X≠Y }    (P3)
```

How to elaborate these properties?

Properties P_1 to P_3 generalize the examples E_1 to E_3, respectively. As in the preceding sample specification, there is no real motivation to explicitly generalize the other examples. But, this time, it is possible to generalize the *nil* constant into a variable without always having to introduce recursive atoms. The resulting allowed properties are:

$$\mathit{firstPlateau}([X \mid T], [X], T) \Leftarrow T{\neq}[X \mid \_] \quad (1)$$

$$\mathit{firstPlateau}([X, Y \mid T], [X, Y], T) \Leftarrow X{=}Y \wedge T{\neq}[Y \mid \_] \quad (2)$$

$$\mathit{firstPlateau}([X, Y \mid T], [X], [Y \mid T]) \Leftarrow X{\neq}Y \quad (3)$$

and the disallowed properties are:

$$\mathit{firstPlateau}([X \mid T], [X \mid U], V) \Leftarrow T{=}[X \mid \_] \wedge \mathit{firstPlateau}(T, U, V)$$

$$\mathit{firstPlateau}([X, Y \mid T], [X, Y \mid U], V) \Leftarrow X{=}Y \wedge T{=}[Y \mid \_] \wedge \mathit{firstPlateau}(T, U, V)$$

Each property (*i*) is more precise than its counterpart P_i, for i=1,2,3. Also, property (1) actually subsumes property (3). So the probably most useful set of properties would be {(1), (2)}. But the property set above should also be convenient for algorithm design, even though it is less precise. The same holds for any set of properties chosen among the six above, as long as the predicates ≠/2 and =/2 occur in it (the latter may actually be hidden in a unification). ♦

Example 6-3: A sample version of *EP(delOddElems)* is:

```
ℰ(delOddElems) = {  delOddElems([],[])                    (E1)
                    delOddElems([0],[0])                  (E2)
                    delOddElems([1],[])                   (E3)
                    delOddElems([2,4],[2,4])              (E4)
                    delOddElems([6,3],[6])                (E5)
                    delOddElems([5,8],[8])                (E6)
                    delOddElems([7,9],[]) }               (E7)
𝒫(delOddElems) = {  delOddElems([X], []) ⇐ odd(X)         (P1)
                    delOddElems([X],[X]) ⇐ ¬odd(X) }      (P2)
```

where *odd*/1 is assumed to be a primitive, and *odd(N)* holds iff *N* is an odd integer. Similar specifications can be elaborated for other filtering problems. ♦

Example 6-4: A sample version of *EP(efface)* is:

```
ℰ(efface) = {  efface(a,[a],[])                           (E1)
               efface(b,[b,c],[c])                        (E2)
               efface(e,[d,e],[d])                        (E3)
               efface(f,[f,g,h],[g,h])                    (E4)
               efface(j,[i,j,k],[i,k])                    (E5)
               efface(p,[m,n,p],[m,n]) }                  (E6)
𝒫(efface) = {  efface(X,[X|T],T)                          (P1)
               efface(X,[Y,X|T],[Y|T]) ⇐ X≠Y }            (P2)
```

Note that, in property P_2, the conclusion actually also holds if X=Y. In other words, property P_2 could be rewritten as follows:

$$\mathit{efface}(X, [Y, X \mid T], [Y \mid T])$$

But this new version no longer contains the predicate ≠/2, which is so crucial for disambiguating the examples. This is a useful illustration of the following observation: properties are implications (even though most of them could be recast as equivalences and still be correct), but nothing prejudices that their conditions must be as general as possible. In other words, properties are meant to be informative, rather than maximally general.

Also note that the following Horn clause:

$$\mathit{efface}(X, [Y \mid T], U) \Leftarrow X{\neq}Y$$

is an incorrect property, because $X{\neq}Y$ doesn't imply that $X{\in}[Y \mid T]$, which is required by the informal specification. The only way to correct this is to add the atom *efface(X,T,U)* to the condition, but the resultant recursive Horn clause is a disallowed property. ♦

Example 6-5: A sample version of *EP(firstN)* is:

```
ℰ(firstN) = {  firstN(0,[],[])                          (E1)
               firstN(0,[a],[])                         (E2)
               firstN(1,[b,c],[b])                      (E3)
               firstN(2,[d,e,f],[d,e])                  (E4)
               firstN(3,[g,h,i,j],[g,h,i]) }            (E5)
𝒫(firstN) = {  firstN(0,T,[])                           (P1)
               firstN(1,[X|T],[X])                      (P2)
               firstN(2,[X,Y|T],[X,Y]) }                (P3)
```

Example 6-6: A sample version of *EP(insert)* is:

```
ℰ(insert) = {  insert(1,[],[1])                         (E1)
               insert(2,[3],[2,3])                      (E2)
               insert(5,[4],[4,5])                      (E3)
               insert(6,[7,8],[6,7,8])                  (E4)
               insert(10,[9,11],[9,10,11])              (E5)
               insert(14,[12,13],[12,13,14]) }          (E6)
𝒫(insert) = {  insert(X,[],[X])                         (P1)
               insert(X,[Y],[Y,X]) ⇐ X>Y                (P2)
               insert(X,[Y|T],[X,Y|T]) ⇐ X≤Y }          (P3)
```

Example 6-7: A sample version of *EP(member)* is:

```
ℰ(member) = {  member(a,[a])                            (E1)
               member(b,[b,c])                          (E2)
               member(c,[b,c])                          (E3)
               member(d,[d,e,f])                        (E4)
               member(e,[d,e,f])                        (E5)
               member(f,[d,e,f]) }                      (E6)
𝒫(member) = {  member(X,[X|T])                          (P1)
               member(X,[Y,X|T]) }                      (P2)
```

Example 6-8: A sample version of *EP(permutation)* is:

```
ℰ(permutation) = {  permutation([],[])                  (E1)
                    permutation([a],[a])                (E2)
                    permutation([b,c],[b,c])            (E3)
                    permutation([b,c],[c,b])            (E4)
                    permutation([d,e,f],[d,e,f])        (E5)
                    permutation([d,e,f],[d,f,e])        (E6)
                    permutation([d,e,f],[e,d,f])        (E7)
                    permutation([d,e,f],[e,f,d])        (E8)
                    permutation([d,e,f],[f,d,e])        (E9)
                    permutation([d,e,f],[f,e,d]) }      (E10)
𝒫(permutation) = {  permutation([X],[X])                (P1)
                    permutation([X,Y],[X,Y])            (P2)
                    permutation([X,Y],[Y,X]) }          (P3)
```

Example 6-9: A sample version of *EP(plateau)* is:

```
ℰ(plateau) = {  plateau(1,b,[b])                        (E1)
                plateau(2,c,[c,c])                      (E2)
                plateau(3,d,[d,d,d]) }                  (E3)
```

```
𝒫(plateau) = {  plateau(1,X,[X])                    (P1)
                plateau(2,X,[X,X]) }                (P2)
```

Example 6-10: A sample version of *EP(sort)* is:

```
ℰ(sort) = { sort([],[])                             (E1)
            sort([1],[1])                           (E2)
            sort([2,3],[2,3])                       (E3)
            sort([3,2],[2,3])                       (E4)
            sort([4,5,6],[4,5,6])                   (E5)
            sort([4,6,5],[4,5,6])                   (E6)
            sort([5,4,6],[4,5,6])                   (E7)
            sort([5,6,4],[4,5,6])                   (E8)
            sort([6,4,5],[4,5,6])                   (E9)
            sort([6,5,4],[4,5,6]) }                 (E10)
𝒫(sort) = { sort([X],[X])                           (P1)
            sort([X,Y],[X,Y]) ⇐ X≤Y                 (P2)
            sort([X,Y],[Y,X]) ⇐ X>Y }               (P3)
```

Example 6-11: A sample version of *EP(sum)* is:

```
ℰ(sum) = {  sum([],0)                               (E1)
            sum([7],7)                              (E2)
            sum([4,5],9)                            (E3)
            sum([2,3,1],6) }                        (E4)
𝒫(sum) = {  sum([X],X)                              (P1)
            sum([X,Y],R) ⇐ add(X,Y,R) }             (P2)
```

6.3 Future Work

Regardless of the actually chosen language for properties, it is quite likely that the latter are some form of a generalization of examples. It would therefore be interesting to investigate more formally how properties can be elaborated from examples. We even conjecture that this process might be partially automated.

6.4 Related Work

In terms of related work, specifications by examples are surveyed in Section 3.1. Also, the notion of specifying property can be traced back to the notion of specifying axiom, and specifications by axioms are surveyed in Section 2.1. In this section, we only survey research on extending example-based specifications by another incomplete information source.

Shapiro pointed out that the oracle of his *Model Inference System* (MIS, see Section 3.4.1) could be partly mechanized by the incorporation of "constraints and partial specifications" [Shapiro 82, page 79]. The idea is investigated by [Lichtenstein and Shapiro 88], whose system asks non-ground queries to an oracle.

Shapiro's idea has also been picked up by [Drabent *et al*. 88]: they define four kinds of *assertions* that may be added to a specification by examples. These assertions describe approximate knowledge about the intended model of the specified program. The language for assertions is Horn clauses plus negation. A *positive assertion* defines a set of atoms that are valid in the intended model. For example:

$$insert(X, T, U) \Leftarrow integer(X) \wedge sorted(T) \wedge sorted(U) \wedge permutation([X \mid T], U)$$

A *negative assertion* defines a set of atoms that are not valid in the intended model. For example:

$$\neg\, sort(T, U) \Leftarrow member(X, T) \wedge \neg\, member(X, U)$$

A *positive existential assertion* defines a set of atoms that are satisfiable in the intended model. For example:

$$sort(T, U) \Leftarrow integerList(T)$$

A *negative existential assertion* defines a set of atoms that are not satisfiable in the intended model. For example:

$$\neg\, sort(T, U) \Leftarrow integerList(U) \wedge \neg\, sorted(U)$$

These assertions are used for partly mechanizing the oracle of Shapiro's MIS. They constitute a possible instantiation of our notion of properties.

Another interesting idea is proposed by [De Raedt and Bruynooghe 92]: in view of cross-fertilization, they represent the problems of intensional knowledge-base updating and incremental concept-learning as particular cases of the more general problem of belief updating from *integrity constraints* and queries. Indeed, intensional knowledge-base updating can benefit from the possibility of asserting non-unit clauses in the knowledge-base. And incremental concept-learning can benefit from the generalization of examples into integrity constraints. The language for integrity constraints is range-restricted functor-free clauses. However, some integrity constraints are slightly different from our properties in the sense that they are not meant to (partially) specify the problem, but rather to verify the behavior of the designed program. This is then reflected in the possible presence of predicate(s) other than the one(s) of the examples in the conclusions of integrity constraints.

Similarly, note that properties are totally different from *background knowledge*, as often used in concept-learning from examples. Indeed, properties partially define the predicate(s) that is (are) incompletely specified by the examples, whereas background information defines predicate(s) that is (are) different from the one(s) found in the examples.

6.5 Conclusion

In this chapter, we have developed an approach for incomplete specifications that is based on the notions of examples and properties.

It requires multiple, ground, single-predicate, relational, positive and negative, pre-synthesis examples that are chosen in a consistent way by a human specifier who knows the intended relation.

The presence of properties (whose actual language is application-specific, and thus left unspecified in this part) is meant to overcome the problems of ambiguity and limited expressive power of examples, while still preserving the virtues of naturalness and conciseness of examples. The predicates used in the properties constitute a partial basis set, and thus a partial conceptual bias for synthesis.

Overall, the specification language is quite expressive and readable. Specifications by examples and properties are usually incomplete, and hence ambiguous, but minimal. There is a danger of internal inconsistency and redundancy in such specifications, though.

This specification approach holds the promise of faster and more dependable synthesis of algorithms than from specifications by examples alone.

7 A Framework for Stepwise Logic Algorithm Synthesis

In this chapter, we develop a theoretical framework for stepwise synthesis of logic algorithms from examples and properties. Thus, in Section 7.1, we elaborate correctness criteria for logic algorithms, and in Section 7.2, we develop comparison criteria for logic algorithms. This provides an adequate structure for the formulation, in Section 7.3, of stepwise logic algorithm synthesis strategies. Future work and related work are discussed in Section 7.4 and Section 7.5, respectively, before drawing some conclusions in Section 7.6.

7.1 Correctness of Logic Algorithms

It is important to measure a logic algorithm against its intentions and its specification, as well as to measure a specification against its intentions. Indeed, such correctness criteria allow us to have a grip on any software engineering activity, be it specification elaboration, manual algorithm construction from informal specifications, semi-automatic algorithm synthesis from formal specifications, algorithm transformation, algorithm verification, and so on.

We here consider three levels of objects, namely the intended relation, the specified relation (by the specification), and the computed relation (by the logic algorithm). This means that we restrict our attention to relations, and not to any kind of intentions. Moreover, we assume that the intended relation is known, even if there only is an informal description of it: no fixed language is imposed for expressing the intentions. So if the intended relation is informally defined, then the correctness criteria cannot be automatically verified. They may however be used to guide software engineering activities. By specifications, we here implicitly mean specifications by examples and properties. This may be generalized to any kind of logic specifications, as in [Deville and Flener 93]. A fourth level, the actually-computed relation (by a logic program) could also be considered here, but we here put the focus exclusively on the logical part of software engineering activities. Comparing these three levels of relations one-by-one gives rise to three sets of correctness criteria. After completing this introduction with more formal definitions of the three levels of relations, this section is divided into three sub-sections, each dedicated to one of the three sets of correctness criteria.

Since we are only concerned with the declarative semantics of logic statements, we define model-theoretic criteria, rather than proof-theoretic ones. The used logic framework is a Herbrand-based first-order logic. This means that we are here only interested in Herbrand interpretations. For simplicity and uniformity, we assume an (infinite) language underlying our specifications and logic algorithms. Some correctness criteria have already been given in Chapter 4, but they do not handle examples and properties. We thus start from scratch for the elaboration of a suitable set of criteria.

Let $\mathcal{R}$ be the n-ary intended relation. The final objective is to obtain a logic program that succeeds on the n-tuples of $\mathcal{R}$, and fails on the n-tuples of $\underline{\mathcal{R}}$ (which is the complement of $\mathcal{R}$ in the set of all ground n-tuples over the universe $\mathcal{U}$ of terms).

We now formally define the notion of specified relation. It consists of the set of n-tuples for which the specified predicate r/n is *true* according to the specification, and of the set of n-tuples for which the specified predicate r/n is *false* according to the specification:

Definition 7-1: The *specified relation* of a specification by examples and properties $EP(r)$ consists of the following two sets:

$$EP^{+}(r) = \{\boldsymbol{t} \mid EP(r) \models r(\boldsymbol{t})\}$$

$$EP^{-}(r) = \{\boldsymbol{t} \mid EP(r) \models \neg\, r(\boldsymbol{t})\}.$$

It is clear that the specified relation is usually different from the intended relation. It is usually expected to be a subset of the intended relation, though. It is also clear that $EP^{-}(r)$ is not always the complement of $EP^{+}(r)$. We however assume that $EP(r)$ is internally consistent, that is that the intersection between $EP^{+}(r)$ and $EP^{-}(r)$ is empty.

A logic algorithm induces a computed relation. As for specifications, it consists of two sets:

Definition 7-2: The *computed relation* of a logic algorithm $LA(r)$ consists of:

$$LA^{+}(r) = \{\boldsymbol{t} \mid LA(r) \models r(\boldsymbol{t})\}$$
$$LA^{-}(r) = \{\boldsymbol{t} \mid LA(r) \models \neg\, r(\boldsymbol{t})\}.$$

Let $LA(r)$ be $r(\boldsymbol{X}) \Leftrightarrow Def[\boldsymbol{X}]$. In the sequel, we assume that *Def* contains only primitive predicates and possibly the *r/n* predicate. This restriction amounts to assuming that the subproblems involved in $LA(r)$ have been—or will be—correctly implemented, and can thus be seen as primitives for $LA(r)$. This restriction is only for ease of understanding, and can be overcome by simultaneously considering $LA(r)$ and its non-primitive predicates, as in [Deville 90].

In order to stress the differences between a specification and a logic algorithm, we further restrict logic algorithms to those designed by structural induction (as in [Deville 90]). This means that some well-founded relation can be defined between the recursive literals and the head of the logic algorithm. This emphasizes the algorithmic aspect of the formula. This also expresses that the obtained logic algorithm terminates (for ground queries). If such a requirement is not fulfilled, then, according to our framework, the logic algorithm is considered a specification.

When a logic algorithm is designed by structural induction on some parameter (as in Chapter 4 for instance), then predicate *r/n* can be interpreted in any Herbrand model of $LA(r)$.

Theorem 7-1: *If LA(r) is designed by structural induction, then the interpretation of r/n is the same in all the Herbrand models of LA(r).*

Proof 7-1: *Base case*. In a design by structural induction of *Def*, there are disjuncts in *Def* that are without recursion. Since all predicates other than *r/n* have a fixed interpretation in all the Herbrand models of $LA(r)$, so does every instance of *r/n* that satisfies the non-recursive disjuncts. *Induction*. Since $LA(r)$ is designed by structural induction, in any disjunct with recursion, the recursive literals involve parameters that are smaller, according to some well-founded relation, than those in the head. More precisely, for every ground instance of the logic algorithm such that the non-recursive literals in the disjunct are *true*, the recursive literals have smaller parameters than the head. Hence, by the induction hypothesis, the recursive instances of *r/n* have a fixed interpretation in all the Herbrand models. Since the other, non-recursive literals also have a fixed interpretation, so does *r/n* for the recursive disjuncts. ❑

The following corollary is then obvious:

Corollary 7-2: *If LA(r) is designed by structural induction, then $LA^{+}(r)$ is the complement of $LA^{-}(r)$.*

In the sequel, we thus only consider recursive logic algorithms where some well-founded relation can be defined between the recursive literals and the head. We thus have to enforce that a synthesis mechanism doesn't design non-terminating recursion (for ground queries).

For convenience, correctness criteria are here defined wrt a Herbrand interpretation $\Im$, called the *intended interpretation*, which is such that the following two conditions hold:

- $r(\boldsymbol{t})$ is *true* in $\Im$ iff $\mathcal{R}(\boldsymbol{t})$ holds,
- $\Im$ is a model of all primitive predicates.

Note that $\Im$ captures the intended relation $\mathcal{R}$, since the interpretation of *r/n* in $\Im$ is $\mathcal{R}$. So $\mathcal{R}$ does not have to be explicitly considered in the correctness criteria.

Example 7-1: Here are four logic algorithms for *sum*/2, as specified in Example 6-11:

$LA_1(sum)$: `sum(L,S)` ⇔ `L=[]` ∧ `S=0`

$LA_2(sum)$: `sum(L,S)` ⇔ `L=[]` ∧ `S=0`
∨ `L=[HL|TL]` ∧ `sum(TL,TS)` ∧ `add(HL,TS,S)`

$LA_3(sum)$: `sum(L,S)` ⇔ `L=[]` ∧ `S=0`
∨ `length(L,N)` ∧ `N>0` ∧ `sub(S,TS,HL)`

$LA_4(sum)$: `sum(L,S)` ⇔ `L=[]` ∧ `S=0`
∨ `L=[HL|TL]`

where *sub(X,Y,D)* holds iff *add(Y,D,X)* holds. The intended relation is denoted *Sum*. We use these logic algorithms in the sequel to illustrate our purpose. ♦

Three levels of correctness criteria can now be identified: measuring a logic algorithm against its intentions, measuring a logic algorithm against its specification, and comparing a specification and its intentions. We discuss these three levels in Section 7.1.1 to Section 7.1.3, respectively.

7.1.1 Logic Algorithms and Intentions

The idea behind correctness of a logic algorithm wrt the intentions is to state that the intended relation $\mathcal{R}$ is identical to the relation computed by *LA(r)*:

$$\mathcal{R} = LA^+(r)$$
$$\overline{\mathcal{R}} = LA^-(r)$$

Correctness thus states an identity, in the Herbrand models of *LA(r)*, between the intended relation $\mathcal{R}$ and the interpretation of predicate *r/n*. The second criterion, which in general is not a consequence of the first one, is necessary to handle logic algorithms with negation (also see [Deville 90]).

Since we only consider logic algorithms that are designed by structural induction, correctness reduces, by Corollary 7-1, to $\mathcal{R} = LA^+(r)$. Moreover, partial correctness is achieved iff $\mathcal{R} \supseteq LA^+(r)$ (that is, iff the atoms "computed" by *LA(r)* are correct), and completeness is achieved iff $\mathcal{R} \subseteq LA^+(r)$ (that is, iff all the correct atoms are "computed" by *LA(r)*). These criteria are in the sequel called the *intuitive criteria*.

The total correctness of a logic algorithm wrt its intended relation can however be re-expressed more conveniently:

Definition 7-3: *LA(r)* is *totally correct wrt* $\mathcal{R}$ iff $r(\boldsymbol{X}) \Leftrightarrow Def[\boldsymbol{X}]$ is *true* in $\Im$.

We now show that this actual criterion is equivalent to its intuitive counterpart:

Theorem 7-3: $r(\boldsymbol{X}) \Leftrightarrow Def[\boldsymbol{X}]$ *is true in* $\Im$ *iff* $\mathcal{R} = LA^+(r)$.

Proof 7-3: Let's prove the involved implications one by one:

(1) Suppose that $r(\boldsymbol{X}) \Leftrightarrow Def[\boldsymbol{X}]$ is *true* in $\Im$. $\Im$ is thus a Herbrand model of *LA(r)*. By Theorem 7-1, the interpretation of *r/n* is $\mathcal{R}$ in all the Herbrand models of *LA(r)*. Hence $\mathcal{R} = LA^+(r)$.

(2) Suppose that $\mathcal{R} = LA^+(r)$. By the definition of $LA^+(r)$, $r(\boldsymbol{t})$ is *true* in all the Herbrand models of *LA(r)* iff $\boldsymbol{t} \in \mathcal{R}$, or, equivalently, iff $r(\boldsymbol{t})$ is *true* in $\Im$. Hence $\Im$ is also a model of *LA(r)*, that is $r(\boldsymbol{X}) \Leftrightarrow Def[\boldsymbol{X}]$ is *true* in $\Im$. ❑

Total correctness is as usual decomposed into partial correctness and completeness:

Definition 7-4: *LA(r)* is *partially correct wrt* $\mathcal{R}$ iff $r(\boldsymbol{X}) \Leftarrow Def[\boldsymbol{X}]$ is *true* in $\Im$.

Definition 7-5: *LA(r)* is *complete wrt* $\mathcal{R}$ iff $r(\boldsymbol{X}) \Rightarrow Def[\boldsymbol{X}]$ is *true* in $\Im$.

Example 7-2: $LA_2(sum)$ is totally correct wrt *Sum*. $LA_1(sum)$ is only partially correct wrt *Sum*. $LA_3(sum)$ and $LA_4(sum)$ are only complete wrt *Sum*.

Let's show that these actual criteria are slightly stronger than their intuitive counterparts.

Theorem 7-4: *Assuming* $Def[\boldsymbol{X}]$ *is in disjunctive normal form and without recursion through negation, the following assertions hold:*

(1) *If* $r(\boldsymbol{X}) \Leftarrow Def[\boldsymbol{X}]$ *is true in* $\Im$, *then* $\mathcal{R} \supseteq LA^+(r)$;

(2) *If* $r(\boldsymbol{X}) \Rightarrow Def[\boldsymbol{X}]$ *is true in* $\Im$, *then* $\mathcal{R} \subseteq LA^+(r)$.

Proof 7-4: We only prove (1). The proof of (2) is analogous. Let's perform a proof by contradiction: suppose that $r(\boldsymbol{X}) \Leftarrow Def[\boldsymbol{X}]$ is *true* in $\Im$, but that $\mathcal{R} \not\supseteq LA^+(r)$, or, equivalently, that $LA^+(r) \setminus \mathcal{R} \neq \varnothing$. Take $\boldsymbol{t}_0 \in LA^+(r) \setminus \mathcal{R}$, with $\boldsymbol{t}_0$ minimal according to the well-founded relation "<" used for the design of *LA(r)*. Such a minimal element exists since "<" is well-founded. Now, since $\boldsymbol{t}_0 \in LA^+(r)$, we have that $r(\boldsymbol{t}_0)$, and thus $Def[\boldsymbol{t}_0]$, are *true* in all the Herbrand models of *LA(r)*. If we can prove (see below) that $Def[\boldsymbol{t}_0]$ is also *true* in $\Im$, then, using the first hypothesis above, $r(\boldsymbol{t}_0)$ is *true* in $\Im$, that is $\boldsymbol{t}_0 \in \mathcal{R}$. This contradicts the fact that $\boldsymbol{t}_0 \in LA^+(r) \setminus \mathcal{R}$. Hence assertion (1) holds.

So let's prove now that $Def[\boldsymbol{t}_0]$ is also *true* in $\Im$. All predicates in $Def[\boldsymbol{t}_0]$ other than *r/n* are necessarily primitive predicates and have a fixed interpretation in all the Herbrand models of *LA(r)*. Hence $\Im$ is equivalent to all the Herbrand models of *LA(r)* as regards the interpretation of these primitive predicates. Thus, if a non-recursive disjunct of $Def[\boldsymbol{t}_0]$ is *true* in all the Herbrand models of *LA(r)*, then it is also *true* in $\Im$. Next, let *D* be some (ground instance of) a recursive disjunct of $Def[\boldsymbol{t}_0]$, with *D* being *true* in all the Herbrand models of *LA(r)*. Since *LA(r)* is designed by structural induction, the recursive atoms are of the form $r(\boldsymbol{t}_1)$, with $\boldsymbol{t}_1$ "<" $\boldsymbol{t}_0$. We also have that $\boldsymbol{t}_1 \in LA^+(r)$, since *D* is *true* in all the Herbrand models of *LA(r)*. In order to have *D* also *true* in $\Im$, it is sufficient to show that $r(\boldsymbol{t}_1)$ is *true* in $\Im$. This is the case because $\boldsymbol{t}_1 \in LA^+(r)$, and $\boldsymbol{t}_1$ "<" $\boldsymbol{t}_0$, and $\boldsymbol{t}_0$ is minimal in $LA^+(r) \setminus \mathcal{R}$. Hence $\boldsymbol{t}_1 \in \mathcal{R}$, and thus $r(\boldsymbol{t}_1)$ is *true* in $\Im$. ❑

Note that Theorem 7-4 does not hold when there is recursion through negation. We show this by constructing adequate counter-examples:

- Regarding partial correctness, let $\mathcal{R}$= {[],[1]}, and let *LA(r)* be:

```
r(L) ⇔    L=[]
        ∨ L=[_,_|_] ∧ ¬r([1])
```

  In the Herbrand models of *LA(r)*, the interpretation of *r/n* is the set {*r(t)* | *t* is any non-singleton list}, and thus $\mathcal{R} \not\supseteq LA^+(r)$. However, $r(L) \Leftarrow Def[L]$ is *true* in $\Im$, as $Def[L]$ is *true* in $\Im$ only for *L*=[], and as $r([])$ is *true* in $\Im$. Assertion (1) thus doesn't hold when there is recursion through negation.

- Regarding completeness, let $\mathcal{R}$ be the set of lists of 1s, and let *LA(r)* be:

```
r(L) ⇔    L=[]
        ∨ L=[0|T] ∧ r(T)
        ∨ L=[1|T] ∧ ¬r([0|T])
```

  No Herbrand model of *LA(r)* contains $r([1])$, and thus $\mathcal{R} \not\subseteq LA^+(r)$. However, $r(L) \Rightarrow Def[L]$ is *true* in $\Im$. Assertion (2) thus doesn't hold when there is recursion through negation.

Also note that the converse assertions of (1) and (2) do not hold. We show this by constructing adequate counter-examples:

- Regarding partial correctness, let $\mathcal{R}$= {[],[1]}, and let *LA(r)* be:

```
r(L) ⇔    L=[]
        ∨ L=[_,_|_] ∧ r([1])
```

The only Herbrand model of *LA(r)* is $\{r([])\}$, and thus $LA^+(r) = \{[]\} \subseteq \mathcal{R}$. Hence *LA(r)* fulfills the intuitive criterion of partial correctness. However, $r(L) \Leftarrow Def[L]$ is *false* in $\Im$, because *Def*[[1,1,1]] is *true* in $\Im$, but *r*([1,1,1]) is *false* in $\Im$. Hence *LA(r)* does not fulfill the actual criterion of partial correctness. The converse of assertion (1) thus doesn't hold.

- Regarding completeness, let $\mathcal{R}$ be the set of lists of 1s, and let *LA(r)* be:

```
r(L) ⇔    L=[]
        ∨ L=[0|T] ∧ r(T)
        ∨ L=[1|T] ∧ r([0|T])
```

$LA^+(r)$ is the set of lists of 0s and 1s, and thus a superset of $\mathcal{R}$. Hence *LA(r)* fulfills the intuitive criterion of completeness. However, $r(L) \Rightarrow Def[L]$ is *false* in $\Im$, because *r*([1]) is *true* in $\Im$, but *Def*[[1]] is *false* in $\Im$. Hence *LA(r)* does not fulfill the actual criterion of completeness. The converse of assertion (2) thus doesn't hold.

These last two counter-examples clearly show why the actual criteria are "better" than their intuitive counterparts: they prevent logic algorithms that are partially correct (respectively complete) in a "bad" way, namely in that they cannot easily be "extended" to totally correct algorithms.

7.1.2 Logic Algorithm and Specification

Next come criteria for measuring a logic algorithm against its specification. Given a set of examples $\mathcal{E}(r) = \mathcal{E}^+(r) \cup \mathcal{E}^-(r)$, a logic algorithm *LA(r)* is complete wrt $\mathcal{E}(r)$ iff the positive examples are contained[9] in the relation computed by *LA(r)*, but the negative examples are not (that is, iff $\mathcal{E}^+(r) \subseteq LA^+(r)$ and $\mathcal{E}^-(r) \subseteq LA^-(r)$). And *LA(r)* is partially correct wrt $\mathcal{E}(r)$ iff the positive examples contain the computed relation (that is, iff $\mathcal{E}^+(r) \supseteq LA^+(r)$; note that it is meaningless to include here the partial correctness of the negative examples, because $\mathcal{E}^+(r)$ is not usually the complement of $\mathcal{E}^-(r)$ in $\mathcal{U}^n$).

Similar criteria can be expressed for a set of properties $\mathcal{P}(r)$. In the above versions, the sets $\mathcal{E}^+(r)$ and $\mathcal{E}^-(r)$ then have to be replaced by the following two sets:

$$\mathcal{P}^+(r) = \{\boldsymbol{t} \mid \mathcal{P}(r) \models r(\boldsymbol{t})\}$$
$$\mathcal{P}^-(r) = \{\boldsymbol{t} \mid \mathcal{P}(r) \models \neg\, r(\boldsymbol{t})\}$$

Again, these criteria are in the sequel called the *intuitive criteria*. Their following formalization is defined in terms of the intended interpretation $\Im$. Although slightly stronger than the intuitive criteria, the following *actual criteria* are more adapted to a framework of logic algorithm synthesis (see [Deville and Flener 93] for a precise account on this subject):

Definition 7-6: *LA(r)* is *complete wrt* $\mathcal{E}(r)$ iff the following two conditions hold:

- $r(\boldsymbol{t}) \in \mathcal{E}^+(r) \Rightarrow Def[\boldsymbol{t}]$ is *true* in $\Im$;
- $\neg\, r(\boldsymbol{t}) \in \mathcal{E}^-(r) \Rightarrow Def[\boldsymbol{t}]$ is *false* in $\Im$.

Definition 7-7: *LA(r)* is *partially correct wrt* $\mathcal{E}(r)$ iff the following condition holds:

- $r(\boldsymbol{t}) \in \mathcal{E}^+(r) \Leftarrow Def[\boldsymbol{t}]$ is *true* in $\Im$.

Definition 7-8: *LA(r)* is *complete wrt* $\mathcal{P}(r)$ iff the following two conditions hold:

- $\mathcal{P}(r) \models r(\boldsymbol{t}) \Rightarrow Def[\boldsymbol{t}]$ is *true* in $\Im$;
- $\mathcal{P}(r) \models \neg\, r(\boldsymbol{t}) \Rightarrow Def[\boldsymbol{t}]$ is *false* in $\Im$.

9. For ease of notation, the set $\mathcal{E}^+(r)$ of positive examples is here also considered to denote the corresponding set of tuples. Similarly for the set $\mathcal{E}^-(r)$ of negative examples.

Definition 7-9: *LA(r)* is *partially correct wrt* $\mathcal{P}(r)$ iff the following condition holds:

- $\mathcal{P}(r) \models r(\boldsymbol{t}) \Leftarrow Def[\boldsymbol{t}]$ is *true* in $\Im$.

Definition 7-10: *LA(r)* is *totally correct wrt* $\mathcal{E}(r)$ (respectively $\mathcal{P}(r)$) iff *LA(r)* is complete and partially correct wrt $\mathcal{E}(r)$ (respectively $\mathcal{P}(r)$).

Any of all these criteria above holds between a logic algorithm *LA(r)* and a specification *EP(r)* iff it holds between *LA(r)* and both the example set and the property set of *EP(r)*.

Total correctness thus means that what is computed (positively or negatively) by the logic algorithm must be consistent with its (part of the) specification. For a tuple $\boldsymbol{t}$ where the considered (part of the) specification is undecidable (that is, where $\boldsymbol{t} \notin EP^+(r)$ and $\boldsymbol{t} \notin EP^-(r)$), $r(\boldsymbol{t})$ can be either *true* or *false* in the logic algorithm.

Example 7-3: $LA_2(sum)$, $LA_3(sum)$, and $LA_4(sum)$ are complete wrt *EP(sum)*.

7.1.3 Specification and Intentions

Finally, there is consistency of a specification wrt the intentions. For instance, consistency of the examples $\mathcal{E}(r)$ wrt the intended relation $\mathcal{R}$ means that the positive examples are contained in $\mathcal{R}$, and that the negative examples are contained in its complement $\overline{\mathcal{R}}$.

Definition 7-11: $\mathcal{E}(r)$ is *consistent with* $\mathcal{R}$ iff the following two conditions hold:

- $r(\boldsymbol{t}) \in \mathcal{E}^+(r) \Rightarrow r(\boldsymbol{t})$ is *true* in $\Im$ (that is, iff $\mathcal{E}^+(r) \subseteq \mathcal{R}$);
- $\neg\, r(\boldsymbol{t}) \in \mathcal{E}^-(r) \Rightarrow r(\boldsymbol{t})$ is *false* in $\Im$ (that is, iff $\mathcal{E}^-(r) \subseteq \overline{\mathcal{R}}$).

Definition 7-12: $\mathcal{P}(r)$ is *consistent with* $\mathcal{R}$ iff the following condition holds:

- $p \in \mathcal{P}(r) \Rightarrow p$ is *true* in $\Im$ (that is, iff $\mathcal{P}^+(r) \subseteq \mathcal{R}$ and $\mathcal{P}^-(r) \subseteq \overline{\mathcal{R}}$).

Definition 7-13: A specification *EP(r)* is *consistent with* $\mathcal{R}$ iff both the example set and the property set of *EP(r)* are consistent with $\mathcal{R}$.

Example 7-4: *EP(sum)* is consistent with $\mathcal{Sum}$.

The specified relation of a consistent specification is thus a subset of the intended relation. Moreover, if *LA(r)* is partially or totally correct wrt $\mathcal{E}(r)$ (respectively $\mathcal{P}(r)$), and $\mathcal{E}(r)$ (respectively $\mathcal{P}(r)$) is consistent with $\mathcal{R}$, then *LA(r)* is partially correct wrt $\mathcal{R}$.

If there is no formal definition of the intended relation $\mathcal{R}$, then some correctness criteria cannot be applied in a formal way. But they can be used to state features and heuristics of a synthesis mechanism.

7.2 Comparison of Logic Algorithms

It is also important to compare logic algorithms for the same intended relation $\mathcal{R}$. Indeed, this is useful in stepwise synthesis to compare intermediate logic algorithms, and to establish progression criteria.

Let $\mathcal{L}(r)$ be the set of all possible logic algorithms of r/n, where the definition parts only involve some fixed set of primitive predicates and the r/n predicate, and where $X_1,\ldots,X_n$ are the distinct variables used in the heads. Let:

- $LA_1(r)$: $r(X_1,\ldots,X_n) \Leftrightarrow Def_1[X_1,\ldots,X_n]$
- $LA_2(r)$: $r(X_1,\ldots,X_n) \Leftrightarrow Def_2[X_1,\ldots,X_n]$

be two logic algorithms in $\mathcal{L}(r)$.

In Section 7.2.1, we define a criterion for comparing logic algorithms in terms of generality. Since verifying this criterion is only semi-decidable, we introduce, in Section 7.2.2, a sound approximation of this criterion, namely syntactic generalization.

7.2.1 Semantic Generalization

Intuitively, $LA_1(r)$ is less general than $LA_2(r)$ iff Def_1 is "less often" true than Def_2. More formally:

Definition 7-14: $LA_1(r)$ is *less general than* $LA_2(r)$ (denoted $LA_1(r) \preceq LA_2(r)$) iff $\forall X_1 \ldots \forall X_n \, (Def_1 \Rightarrow Def_2)$ is *true*.

The fact of being *more general* ($\succeq$) is defined dually. Two logic algorithms, each more general than the other, are *equivalent* ($\cong$). We use $\prec$ for $\preceq$ and $\not\cong$.

Example 7-5: We have $LA_1(sum) \prec LA_2(sum) \prec LA_3(sum) \prec LA_4(sum)$.

Note that a generalization relationship between two logic algorithms does not amount to a logical implication relation between them.

The set $\mathcal{L}(r)$ modulo $\cong$ (denoted $\mathcal{L}(r)^{\cong}$) is partially ordered under $\preceq$. It includes as least element $\bot_r$ (defined as $r(X_1,\ldots,X_n) \Leftrightarrow false$, and called *bottom*) and as greatest element $\top_r$ (defined as $r(X_1,\ldots,X_n) \Leftrightarrow true$, and called *top*). In order to have an upper bound to any ascending sequence of logic algorithms, let's extend $\mathcal{L}(r)$ to $\mathcal{M}(r)$ by allowing an infinite number of literals in the body of a logic algorithm. We obviously have that $(\mathcal{M}(r)^{\cong}, \preceq)$ is isomorphic to $(\mathcal{P}(\mathcal{U}^n), \subseteq)$, where $\mathcal{P}(\mathcal{S})$ denotes the set of subsets of set $\mathcal{S}$. Hence $(\mathcal{M}(r)^{\cong}, \preceq)$ is also a complete lattice, whose lub operator is the logical *or* connective (denoted $\vee$), and whose glb operator is the logical *and* connective (denoted $\wedge$) over the bodies of logic algorithms.

Comparing logic algorithms in terms of semantic generality can be a difficult task, and is only semi-decidable anyway. We thus define a particular case of this generality relation in terms of purely syntactic criteria.

7.2.2 Syntactic Generalization

In view of defining syntactic criteria for generalization, we need to constrain the logic algorithm language. Thus, in this section, we assume that logic algorithms have bodies that are disjunctions of conjunctions of literals. We represent formulas by multisets, so as to avoid ordering problems. A similar development, though for second-order expressions, but without negation, has been made by [Tinkham 90].

Definition 7-15: Let F be a conjunction of literals (respectively a disjunction of conjunctions of literals). Then $elems(F)$ is $\varnothing$ iff F is the predicate *true* (respectively *false*), and the multiset of literals of F (respectively the multiset of the conjunctions of literals of F), otherwise.

Let's first define syntactic generalization over conjunctions of literals, and then over logic algorithms:

Definition 7-16: A conjunction C_1 is *syntactically less general than* a conjunction C_2 (denoted $C_1 \ll C_2$) *with* a substitution θ iff $elems(C_2\theta) \subseteq elems(C_1)$.

Example 7-6: $p(a,X) \wedge q(Y) \ll p(V,W)$ with $\{V/a, W/X\}$.

Note that this definition is different from classical θ-subsumption for clauses [Plotkin 70], because the used substitution is here given, rather than only constrained to exist. This variant is useful if one wants to compare sets of clauses, as captured in the next definition.

Definition 7-17: $LA_1(r)$ is *syntactically less general than* $LA_2(r)$ (denoted $LA_1(r) \ll LA_2(r)$) iff there is a total function *fct* from $elems(Def_1)$ to $elems(Def_2)$, such that, for every disjunct D in $elems(Def_1)$, there is a substitution θ that only binds existential variables of $LA_2(r)$, such that $D \ll fct(D)$ with substitution θ.

Note that θ cannot bind a universal variable of $LA_2(r)$ since this could actually lead to a specialization. For instance: $r(X,Y) \Leftrightarrow q(Z) \quad > \quad r(X,Y) \Leftrightarrow q(X)$.

Example 7-7: We have $LA_1(sum) \ll LA_2(sum) \ll LA_4(sum)$. However, $LA_2(sum)$ and $LA_3(sum)$ are incomparable under «, as they involve different predicates.

The fact of being *syntactically more general* (») is defined dually. Two logic algorithms, each syntactically more general than the other, are *syntactically equivalent* (≈). Note that syntactical equivalence is more general than alphabetic variance, because of the irrelevance of the ordering of disjuncts within logic algorithms, and of literals within disjuncts.

The set $\mathcal{L}(r)^{\approx}$ is partially ordered under «. The following proposition is a direct consequence of the definitions:

Proposition 7-5: *The relations «, », and ≈ are sub-relations of ≤, ≥, and ≅, respectively.*

We now define an atomic refinement operator, after making two preliminary observations. A *most general literal* in a disjunct D of $LA(r)$ is of the form $p(Z_1,\ldots,Z_m)$ or $\neg p(Z_1,\ldots,Z_m)$, where p is an m-ary predicate, and $Z_1,\ldots,Z_m$ are existential variables occurring exactly once in D. And a *most general term* in a disjunct D of $LA(r)$ is of the form $f(Z_1,\ldots,Z_m)$, where f is an m-ary functor, and $Z_1,\ldots,Z_m$ are existential variables occurring exactly once in D.

Definition 7-18: Let *gen* be a *refinement operator* such that $LA_2(r) \in gen(LA_1(r))$ iff exactly one of the following holds:

- $LA_2(r)$ is derived from $LA_1(r)$ by adding a disjunct to $LA_1(r)$; [10]
- $LA_2(r)$ is derived from $LA_1(r)$ by replacing a disjunct D_1 by a disjunct D_2, such that:
  - D_2 is D_1 without a most-general literal in D_1; [11]
  - D_2 is D_1 where one or more occurrences of a variable V are replaced by a new existential variable W;
  - D_2 is D_1 where one or more occurrences of a most general term in D_1 are replaced by a new existential variable W.

The ability to add a disjunct of course often overrides the need to modify a disjunct, as it suffices to add the modified disjunct in the first place, for instance when creating a logic algorithm from $\bot_r$. However, this is not always possible, for instance when modifying an existing logic algorithm into another one.

Example 7-8: $LA_2(sum) \in gen(LA_1(sum))$, and $LA_4(sum) \in gen^3(LA_2(sum))$.

Let us now relate the refinement operator *gen* to the generality relation «:

Theorem 7-6: *The following three assertions hold*:

(1) *gen is a syntactic generalization operator*:
$\forall LA'(r) \in gen(LA(r)) \quad LA(r) \ll LA'(r)$;

(2) *gen can generate any syntactic generalization*:
$LA_1(r) \ll LA_2(r) \Leftrightarrow \exists n\, \exists LA_2'(r) \in gen^n(LA_1(r)) \quad LA_2'(r) \approx LA_2(r)$;

(3) *gen can generate all logic algorithms of $\mathcal{L}(r)$ from $\bot_r$*:
$gen^*(\bot_r) = \mathcal{L}(r)$.

Proof 7-6: Analogous to the proofs in [Tinkham 90]. ❑

An inverse operator *spec* of *gen* is similarly defined, such that *spec* is a syntactic specialization operator that can generate all logic algorithms of $\mathcal{L}(r)$ from $\top_r$.

10. By convention, adding a disjunct D to $\bot_r$ amounts to replacing *false* by D.
11. By convention, deleting the unique literal of a singleton disjunct D amounts to replacing D by *true* if there is no *true* disjunct yet in $LA_1(r)$, and to discarding D, otherwise.

7.3 Stepwise Logic Algorithm Synthesis Strategies

It is interesting to decompose a synthesis process into a series of steps, each designing an intermediate logic algorithm. Indeed, this:

- allows different methods to be deployed at each step, thus enforcing a neat separation of concerns;
- yields monitoring points where correctness and comparison criteria can be applied, hence measuring the effectiveness and progression of synthesis.

Stepwise synthesis can be:

- *incremental*: examples or properties are presented one-by-one, each presentation yielding a run through all synthesis steps;
- *non-incremental*: examples or properties are presented all-at-once, yielding a single run through all synthesis steps;
- *increasing*: each intermediate logic algorithm is more general than its predecessor;
- *decreasing*: each intermediate logic algorithm is less general than its predecessor;
- *monotonic*: synthesis is either increasing, or decreasing;
- *non-monotonic*: synthesis is neither increasing, nor decreasing;
- *consistent*: each intermediate logic algorithm is complete wrt the examples and properties presented so far;
- *inconsistent*: each intermediate logic algorithm is not necessarily complete wrt the examples and properties presented so far.

Using the criteria defined in the two previous sections, a huge variety of stepwise synthesis strategies can be defined, fitting any valid combination of the features enumerated above. In Section 7.3.1, we briefly sketch a strategy for incremental synthesis, and in Section 7.3.2, we fully describe a strategy for non-incremental synthesis.

7.3.1 An Incremental Synthesis Strategy

In the case of incremental synthesis, let's view the steps of one synthesis increment as a macro-step performing a transformation *trans*. Synthesis is then the design of a series of logic algorithms:

$$LA_0(r), LA_1(r), \ldots, LA_i(r), \ldots$$

from a series of specifications $\mathcal{S}_i(r)$ that are sets of examples and properties:

$$\mathcal{S}_1(r), \ldots, \mathcal{S}_i(r), \ldots$$

with:

$$\mathcal{S}_i(r) \subseteq \mathcal{S}_{i+1}(r), \text{ for } i \geq 0$$

such that the following two conditions hold:

- $LA_0(r) = \bot_r$,
- $LA_i(r) = trans(LA_{i-1}(r), \mathcal{S}_i(r))$, for $i > 0$.

This covers *iterative synthesis*, where only the last presented example or property is actually used by *trans*. If *trans* is monotonic and continuous (wrt the $\leq$ order on logic algorithms), then $trans^{\omega}(\bot_r)$ is its least fixpoint. So if *trans* preserves partial correctness wrt $\mathcal{R}$, then the fixed point is also partially correct wrt $\mathcal{R}$. Note that completeness wrt $\mathcal{R}$ is not necessarily achieved, and that the resulting logic algorithm can involve infinitely many literals. This incremental strategy is increasing (hence monotonic); it may be consistent or inconsistent.

7.3.2 A Non-Incremental Synthesis Strategy

Let's first give a criterion for upward (or partial-correctness preserving) progression:

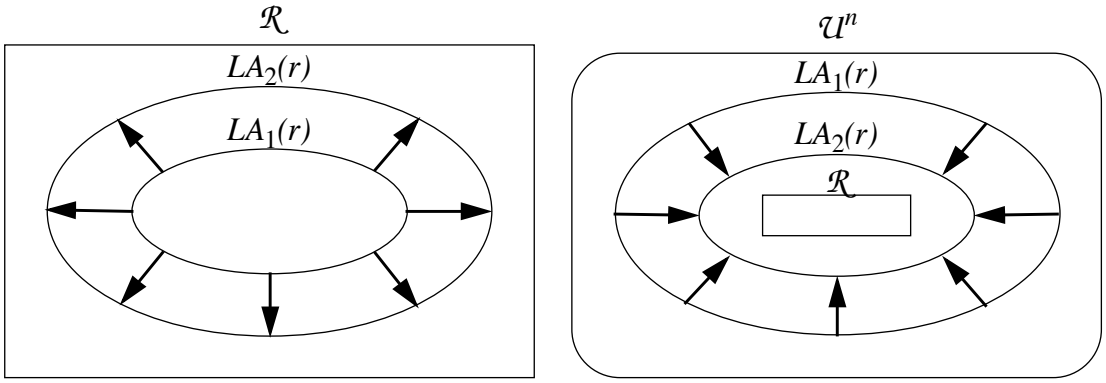


Figure 7-1: (a) Upward and (b) downward progression

Definition 7-19: (See Figure 7-1a.) If the following two conditions hold:
- $LA_2(r) \geq LA_1(r)$,
- $LA_2(r)$ is partially correct wrt $\mathcal{R}$,

then $LA_2(r)$ is a *better partially correct approximation of* $\mathcal{R}$ *than* $LA_1(r)$.

Dually, there is a criterion for downward (or completeness preserving) progression:

Definition 7-20: (See Figure 7-1b.) If the following two conditions hold:
- $LA_2(r) \leq LA_1(r)$,
- $LA_2(r)$ is complete wrt $\mathcal{R}$,

then $LA_2(r)$ is a *better complete approximation of* $\mathcal{R}$ *than* $LA_1(r)$.

Example 7-9: $LA_3(sum)$ is a better complete approximation of $\mathcal{S}um$ than $LA_4(sum)$.

A first idea of a non-incremental stepwise synthesis strategy (with a fixed, finite number f of predefined steps) is to achieve downward progression:

At **Step 1**, "create" $LA_1(r)$ such that:
- $LA_1(r)$ is complete wrt $\mathcal{R}$.

At **Step *i*** ($2 \leq i \leq f$), transform $LA_{i-1}(r)$ into $LA_i(r)$ such that:
- $LA_i(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_{i-1}(r)$.

An obvious way to do Step 1 is to make $LA_1(r)$ equal to T_r.

But this strategy doesn't take care of the partial-correctness aspects. We thus want to define a specialization operator that allows the transformation of the series of intermediate logic algorithms into another series of intermediate logic algorithms that actually reflects upward progression. The idea is to expand the disjuncts of a logic algorithm with equality atoms, each such atom unifying one of the variables of a disjunct with some constant(s) obtained by "evaluating", in $\Im$, that disjunct on some example of the specification.

Example 7-10: Let $LA_5(sum)$ be as follows:

```
sum(L,S) ⇔
    L=[]
  ∨ L=[HL|TL] ∧ sum(TL,TS)
```

Its expansion according to the examples of *EP(sum)*, as in Example 6-11, is as follows:

```
sum(L,S) ⇔
    L=[]        ∧   L=[] ∧ S=0
  ∨ L=[HL|TL]   ∧ sum(TL,TS)
                ∧   L=[7] ∧ HL=7 ∧ TL=[] ∧ TS=0 ∧ S=7
                  ∨ L=[4,5] ∧ HL=4 ∧ TL=[5] ∧ TS=5 ∧ S=9
                  ∨ L=[2,3,1] ∧ HL=2 ∧ TL=[3,1] ∧ TS=4 ∧ S=6
```

where the boldface atoms are synthesized atoms, and the others are trailing atoms. ♦

The series of expanded logic algorithms can be shown to progress upwards (see below). To achieve this expansion, we here only consider logic algorithms whose bodies are disjunctions of conjunctions of literals. The used predicates are assumed to be either primitives or the predicate *r/n* that is defined by the logic algorithm. Let's start with a few basic definitions.

Definition 7-21: Let $D[\boldsymbol{X}]$ be a disjunct of a logic algorithm *LA(r)*, such that the only free variables of *D* are the *n* universal variables $\boldsymbol{X}$ in the head of *LA(r)*. We say that $D[\boldsymbol{X}]$ *covers* atom $r(\boldsymbol{t})$ iff $\exists D[\boldsymbol{t}]$ is *true* in $\Im$.

Example 7-11: The first disjunct of $LA_5(sum)$ covers the example *sum*([],0). Its second disjunct covers the example *sum*([7],7), but it doesn't cover the example *sum*([],0).

Definition 7-22: Let disjunct $D[\boldsymbol{X}]$ of a logic algorithm *LA(r)* cover atom $r(\boldsymbol{t})$. The *expansion of* $D[\boldsymbol{X}]$ *wrt* $r(\boldsymbol{t})$ is $D[\boldsymbol{X}] \wedge \boldsymbol{X}=\boldsymbol{t} \wedge wit(\boldsymbol{Y})$, where $\boldsymbol{Y}$ are the existential variables of $D[\boldsymbol{X}]$, and $wit(\boldsymbol{Y})$ is a formula that is *true* in $\Im$ only for (ground) witnesses $\boldsymbol{w}$ of $\boldsymbol{Y}$ in $\exists D[\boldsymbol{t}]$ (that is, $D[\boldsymbol{t}]\{\boldsymbol{Y}/\boldsymbol{w}\}$ is *true* in $\Im$ iff $wit(\boldsymbol{w})$ is *true* in $\Im$).

Example 7-12: The expansion of the second disjunct of $LA_5(sum)$ wrt *sum*([7],7) is:

$$L=[HL|TL] \wedge sum(TL,TS) \wedge (L=[7] \wedge S=7) \wedge (HL=7 \wedge TL=[] \wedge TS=0)$$

The expansion of the second disjunct of $LA_5(sum)$ wrt *sum*([1,2,*A*],6) is:

$$L=[HL|TL] \wedge sum(TL,TS) \wedge$$
$$(L=[1,2,A] \wedge S=6) \wedge (HL=1 \wedge TL=[2,A] \wedge TS=B \wedge add(2,A,B)) \quad ♦$$

Obviously, $wit(\boldsymbol{Y})$ can always be expressed as a possibly infinite disjunction of equality atoms. In the sequel, we here only consider the situation where $wit(\boldsymbol{Y})$ can be expressed as a finite disjunction of equality atoms (whose left-hand sides are $\boldsymbol{Y}$). Moreover, we then rewrite $wit(\boldsymbol{Y})$ as $\boldsymbol{Y}=\boldsymbol{w}$ (iff $wit(\boldsymbol{Y})$ is a single equality atom whose right-hand side is the term-tuple $\boldsymbol{w}$), or as $\boldsymbol{Y}\in\underline{\boldsymbol{w}}$ (iff $wit(\boldsymbol{Y})$ is a disjunction of equality atoms whose right-hand sides are the term-tuples $\boldsymbol{w}_i$). We call $\underline{\boldsymbol{w}}$ the *witnesses* of $\boldsymbol{Y}$, even though they are not necessarily ground.

For the non-recursive literals, since they are assumed to be primitives, it is possible to compute the witnesses $\underline{\boldsymbol{w}}$ without knowing $\mathcal{R}$. We thus only need an oracle for the specified predicate *r/n*. A candidate mechanized oracle is one that performs deduction using the specification *EP(r)* as knowledge about *r/n*. Such a *deductive oracle* is sound[12] provided *EP(r)* is consistent with $\mathcal{R}$. Other oracles can be imagined, performing analogical reasoning, say. In the sequel we just assume the existence of such an oracle, but do not require it to be sound or complete. A more practical version of the previous definition is thus:

Definition 7-23: Let *O(r)* be an oracle for predicate *r/n*. Let disjunct $D[\boldsymbol{X}]$ of a logic algorithm *LA(r)* cover atom $r(\boldsymbol{t})$. The *expansion of* $D[\boldsymbol{X}]$ *wrt* $r(\boldsymbol{t})$ *and O(r)* is $D[\boldsymbol{X}] \wedge \boldsymbol{X}=\boldsymbol{t} \wedge \boldsymbol{Y}\in\underline{\boldsymbol{w}}$, where $\boldsymbol{Y}$ are the existential variables of $D[\boldsymbol{X}]$, and $\underline{\boldsymbol{w}}$ are the witnesses of $\boldsymbol{Y}$, using *O(r)*, of $\exists D[\boldsymbol{t}]$.

Example 7-13: Take *EP(permutation)* as in Example 6-8, and let *LA(permutation)* be:

```
permutation(L,P) ⇔
    L=[]
  ∨ L=[HL|TL] ∧ permutation(TL,TP)
```

The expansion of the second disjunct of *LA(permutation)* wrt *permutation*([*a*,*b*,*c*],[*c*,*a*,*b*]) and the deductive oracle based on *EP(permutation)* is:

$$L=[HL|TL] \wedge permutation(TL,TP) \wedge$$
$$(L=[a,b,c] \wedge P=[c,a,b]) \wedge (HL=a \wedge TL=[b,c] \wedge TP\in\{[b,c],[c,b]\})$$

12. An oracle is *sound* iff all its answers are correct. An oracle is *complete* iff it gives all correct answers.

If *TL*=[*b*,*c*], then the deductive oracle infers that either *TP*=[*b*,*c*] or *TP*=[*c*,*b*], according to properties P_2 and P_3, respectively. ♦

Example 7-14: Take *EP(efface)* as in Example 6-4, and let *LA(efface)* be:

```
efface(E,L,R) ⇔
    R=[]
  ∨ R=[HR|TR] ∧ efface(TE,TL,TR)
```

The expansion of the second disjunct of *LA(efface)* wrt *efface*(*b*,[*b*,*c*],[*c*]) and the deductive oracle based on *EP(efface)* is:

$$R=[HR|TR] \wedge efface(TE,TL,TR) \wedge$$
$$(E=b \wedge L=[b,c] \wedge R=[c]) \wedge (HR=c \wedge TR=[] \wedge \langle TE,TL\rangle \in \{\langle a,[a]\rangle,\langle X,[X]\rangle\})$$

Indeed, if *TR*=[], then the deductive oracle infers that either <*TE*,*TL*>=<*a*,[*a*]> or <*TE*,*TL*>=<*X*,[*X*]>, according to example E_1 and property P_1, respectively. This illustrates three things. First, some of the existential variables may have to be regrouped into tuples in order to avoid losing some relationships. Second, the availability of the examples to the oracle provokes the appearance of an answer that is totally unrelated to any computation of *efface*(*b*,[*b*,*c*], [*c*]), namely <*TE*,*TL*>=<*a*,[*a*]>. We explain in Section 12.4.2 how such freak answers may be eliminated. Third, variables may appear in answers given by the oracle, such as in <*TE*,*TL*>=<*X*,[*X*]>. It is important to realize that these are existential variables. ♦

Let's now extend the previous definition to entire sets of covered atoms:

Definition 7-24: Let *O(r)* be an oracle for predicate *r*/*n*. Let disjunct $D[\boldsymbol{X}]$ of a logic algorithm *LA(r)* cover all the atoms of an atom set $\mathcal{A}(r) = \{r(\boldsymbol{t}_i)\}$. The *expansion of* $D[\boldsymbol{X}]$ *wrt* $\mathcal{A}(r)$ *and O(r)* is $D[\boldsymbol{X}] \wedge \vee_i (\boldsymbol{X}=\boldsymbol{t}_i \wedge \boldsymbol{Y} \in \underline{\boldsymbol{w}}_i)$, where $\boldsymbol{Y}$ are the existential variables of $D[\boldsymbol{X}]$, and $\underline{\boldsymbol{w}}_i$ are the witnesses of $\boldsymbol{Y}$, using *O(r)*, of $\exists D[\boldsymbol{t}_i]$.

Finally, let's extend this definition to the expansion of an entire logic algorithm:

Definition 7-25: Given an atom set $\mathcal{A}(r)$, an oracle *O(r)*, and a logic algorithm *LA(r)*, the operator *exp*/3 is a total function into $\mathcal{L}(r)$, such that *exp(LA(r), A(r), O(r))* is *LA(r)* where each disjunct has been replaced by its expansion wrt the largest subset of $\mathcal{A}(r)$ that it covers and *O(r)*.

Example 7-15: We effectively have that *exp*(LA_5*(sum),*$\mathcal{E}^+$*(sum),EP(sum))* is as depicted in Example 7-10. ♦

It is obvious that *exp*/3 can be expressed as a sequence of applications of *spec*. Hence *exp* is a syntactic specialization function: *exp(LA(r))* « *LA(r)*, thus *exp(LA(r))* ≤ *LA(r)*.

In the sequel, once a specification *EP(r)* has been clearly mentioned, we assume that *O(r)* is the deductive oracle based on *EP(r)*, and that $\mathcal{A}(r)$ is the positive example set of *EP(r)*. This allows a more compact notation for the *exp*/3 operator.

Definition 7-26: In an expanded logic algorithm, the literals can be partitioned into two categories: the *synthesized literals* are those that are already present before the expansion process, and the *trailing atoms* are the atoms added by the expansion process.

For syntactic brevity, the trailing atoms of a logic algorithm are often summarized into an annotation to the disjunct of synthesized literals they are logically attached to. Such an annotation consists of a set of identifiers of positive examples.

Example 7-16: For instance, *exp*(LA_5*(sum))* could also be written as:

```
sum(L,S) ⇔
    L=[]                                          {E1}
  ∨ L=[HL|TL] ∧ sum(TL,TS)                        {E2,E3,E4}
```

The strategy above can now be refined as follows, so as to have simultaneous upward and downward progressions:

At **Step 1**, "create" $LA_1(r)$ such that:

- $LA_1(r)$ is complete wrt $\mathcal{R}$,
- $exp(LA_1(r))$ is partially correct wrt $\mathcal{R}$.

At **Step** i $(2 \le i \le f)$, transform $LA_{i-1}(r)$ into $LA_i(r)$ such that:

- $LA_i(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_{i-1}(r)$,
- $exp(LA_i(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_{i-1}(r))$.

This strategy is very general.

We now state and prove a generic theorem showing how steps 2 to f of the generic strategy above can be refined in order to yield a practical framework.

Theorem 7-7: *Let* $LA(r)$ *be* $r(\boldsymbol{X}) \Leftrightarrow \vee_{1 \le j \le m} A_j$ *and* $LA'(r)$ *be* $r(\boldsymbol{X}) \Leftrightarrow \vee_{1 \le j \le m} A_j \wedge B_j$, *where* A_j, B_j *are any formulas. The following two assertions hold:*

(1) *If* $LA(r)$ *is complete wrt* $\mathcal{R}$ *and* $\mathcal{R}(\boldsymbol{X}) \wedge A_j \Rightarrow B_j$ $(1 \le j \le m)$
then $LA'(r)$ *is a better complete approximation of* $\mathcal{R}$ *than* $LA(r)$.

(2) *If* $LA(r)$ *is partially correct wrt* $\mathcal{R}$ *and* $A_j \Rightarrow B_j$ $(1 \le j \le m)$
then $LA'(r)$ *is a better partially correct approximation of* $\mathcal{R}$ *than* $LA(r)$.

Proof 7-7: Let's prove these assertions one by one:

(1) By Definition 7-5, the first hypothesis reads:

$$r(\boldsymbol{X}) \Rightarrow \vee_{1 \le j \le m} A_j.$$

By definition of $\Im$, the second hypothesis reads $(1 \le j \le m)$:

$$r(\boldsymbol{X}) \wedge A_j \Rightarrow B_j, \text{ or, equivalently: } r(\boldsymbol{X}) \Rightarrow \neg A_j \vee B_j$$

Combined with the first hypothesis, we get:

$$r(\boldsymbol{X}) \Rightarrow (\vee_{1 \le j \le m} A_j) \wedge (\wedge_{1 \le j \le m} \neg A_j \vee B_j)$$

The right-hand side can be rearranged as:

$$\vee_{1 \le j \le m} A_j \wedge B_j \wedge C_j$$

where C_j is a formula involving $\neg A_i$ and B_i $(1 \le i \le m, i \ne j)$. Hence:

$$r(\boldsymbol{X}) \Rightarrow \vee_{1 \le j \le m} A_j \wedge B_j$$

that is: $LA'(r)$ is complete wrt $\mathcal{R}$. By construction, we have: $LA'(r) \ll LA(r)$, that is, by Theorem 7-5: $LA'(r) \le LA(r)$. Thus: $LA'(r)$ is a better complete approximation of $\mathcal{R}$ than $LA(r)$.

(2) Obviously, we have:

$$\vee_{1 \le j \le m} A_j \wedge B_j \Rightarrow \vee_{1 \le j \le m} A_j$$

Moreover, the second hypothesis implies:

$$\vee_{1 \le j \le m} A_j \Rightarrow \vee_{1 \le j \le m} A_j \wedge B_j$$

Thus: $LA'(r) \cong LA(r)$, that is, in particular: $LA'(r) \ge LA(r)$. Using the first hypothesis, we obtain: $LA'(r)$ is partially correct wrt $\mathcal{R}$. Thus: $LA'(r)$ is a better partially correct approximation of $\mathcal{R}$ than $LA(r)$. ❑

The second hypothesis of assertion (2) ensures then that the introduced literals are redundant with the already existing ones. In other words, as the proof has shown, we then actually have $LA'(r) \cong LA(r)$. The second hypothesis of assertion (1) ensures that the introduced literals are "redundant" with the intended relation $\mathcal{R}$.

In practice, assertion (1) is applied to the logic algorithms $LA_i(r)$, whereas assertion (2) is applied to the logic algorithms $exp(LA_i(r))$, where $i > 1$. The first hypotheses of both assertions need not be proved if they are established by Step 1 and then preserved by application of

Theorem 7-7 to all previous steps. Proving the second condition of assertion (1) can't be done in a formal way for lack of a formal definition of $\mathcal{R}$. However, this can be used to guide a synthesis mechanism, for instance by means of interaction with the specifier, hence increasing the confidence in the synthesis.

Also note that the $exp(LA_i(r))$ actually do not strictly progress upwards at all, and that there is hence no possible upwards convergence. This is not a disaster, because the $LA_i(r)$ may actually strictly progress downwards, and hence possibly converge to a correct logic algorithm. But the $exp(LA_i(r))$ do not regress either, and may be used to show that completeness wrt the examples is preserved (provided this is initially achieved by Step 1). Moreover, the $exp(LA_i(r))$ are useful in that they provide the input data for steps i+1.

7.4 Future Work

This chapter doesn't address *learnability*. This issue is very important for inductive inference, and has mainly been studied for incremental approaches. The definition of the class of algorithms that can be identified from examples and properties, and the characterization of how this identification can be attained (finitely, in-the-limit,...), are open questions so far.

Regarding *generalization models*, the main focus here is on syntactic generalization, because of its usage and adequacy in later chapters. But our framework of comparing logic algorithms in terms of generality can of course be further generalized by incorporating stronger, semantic generalization criteria, such as those of [Buntine 88], which exploit background knowledge.

7.5 Related Work

The here presented criteria for *correctness* of logic algorithms are a direct consequence of the results by [Deville 90], which are also outlined in Chapter 4. A comprehensive survey of correctness criteria for logic programs can be found in the background sections of chapters 4 and 8 of [Deville 90]. Some researchers of the machine learning community, and almost all authors of the ILP community (see Chapter 3), have addressed the correctness issue. The resulting criteria are specific to the chosen specification languages, but very similar (also to ours) once the differences in terminology are abstracted away. A consensus is about to emerge, though. These works usually also address learnability issues.

Many artificial intelligence tasks involve a search space of hypotheses. A *model of generalization* of hypotheses then comes in handy to organize that search space. This allows a more intelligent search than pure enumeration, for instance by pruning uninteresting branches of the search space.

One of the first studies on syntactic generalization was made by [Plotkin 70, 71]. His generalization criterion for clauses, known as θ-*subsumption*, is that clause C θ-subsumes clause D iff there exists a substitution σ such that $C\sigma \subseteq D$. In other words, only the dropping of conditions and the conversion of constants into variables are covered by θ-subsumption. But this criterion is purely syntactic, and thus doesn't exploit possibly existent background knowledge. So he developed another criterion, called *relative subsumption*, which says that clause C generalizes clause D relative to the set of clauses P iff there exists a substitution σ such that $P \models \forall(C\sigma \Rightarrow D)$. He also introduced an induction mechanism for computing the *relative least general generalization (rlgg)* of two clauses relative some other clauses. His search for an *rlgg* goes from specific to general. The drawback of this technique is that, in general, the *rlgg* of two clauses is not finite. This severely restricts the applicability of these results.

The concept of θ-subsumption has been picked up again by [Shapiro 82], who only considers Horn clauses. His search for hypotheses goes from general to specific, using a *most*

general specialization operator to explore a specialization hierarchy. The technique has been applied to logic program synthesis and debugging: specialization of a too general program occurs by discarding too general clauses, and generalization of a too specific program occurs by adding specializations of previously discarded clauses.

This *Model Inference System* (MIS, see Section 3.4.1) has spawned a lot of renewed interest in generalization models. For instance, [Tinkham 90] extends Shapiro's results to a second-order search space. Simultaneously, but independently, [Gegg-Harrison 89, 93] also suggests generalization operators within a second-order logic, though for an entirely different application.

But these models of syntactic generalization are insufficient (because many "wanted" generalizations are not covered), and inadequate (because they cover many "unwanted" generalizations). [Buntine 88] introduces *generalized subsumption* as a stronger model of semantic generalization, of which θ-subsumption is a particular case. Background knowledge is used for inferring more "interesting" generalizations. As we don't exploit such background knowledge in the sequel, these stronger models have not been further investigated here: please refer to [Buntine 88] for more details, for a comparison with other generalization models and logical implication, and for applications for inductive inference and redundancy control.

Building upon the "limited success due to the complexity of constructed clauses" of Buntine's proposal to circumvent Plotkin's negative result, [Muggleton 91] introduces another model of generalization, with more restrictions on hypotheses.

Regarding *stepwise synthesis*, the immense majority of research goes into incremental approaches, with much focus on monotonic and consistent techniques. Other incremental approaches are, for instance, non-monotonic synthesis [Jantke 91], and inconsistent synthesis [Lange and Wiehagen 91].

The non-incremental, stepwise synthesis strategy presented in Section 7.3.2 features the same idea as the incremental *version spaces* strategy of [Mitchell 81], namely simultaneous bottom-up and top-down search, but is after all totally different from it.

7.6 Conclusion

In this chapter, we have developed a general framework for stepwise synthesis of logic algorithms from specifications by examples and properties. It includes correctness criteria for relating the intended, specified, and computed relations, as well as comparison criteria for relating logic algorithms in terms of their generality. As part of a framework, these sets of criteria are modular, as new criteria can be added. The main issue is that such criteria can be used to state strategies of stepwise synthesis, be they incremental or not, monotonic or not, consistent or not. A particular, non-incremental, monotonic, and consistent strategy has been developed in greater detail for use in the sequel.

8 Algorithm Analysis and Algorithm Schemata

Algorithm schemata have become a popular research topic. In Section 8.1, we introduce algorithm schemata and argue that they are an important support for guiding algorithm design. Then, in Section 8.2, we introduce a logic algorithm schema reflecting a divide-and-conquer design strategy. This allows us, in Section 8.3, to refine the framework of Chapter 7 to stepwise and schema-guided synthesis of logic algorithms from examples and properties. Future work and related work are discussed in Section 8.4 and Section 8.5, respectively, before drawing some conclusions in Section 8.6.

8.1 Introduction to Algorithm Schemata

Algorithms can be classified according to their design strategies, such as divide-and-conquer, generate-and-test, global search, local search, top-down decomposition, and so on. Informally, an *algorithm schema* is a template algorithm with a fixed control flow, but without specific indications about the parameters or the actual computations. An algorithm schema thus represents a whole family of particular algorithms that can be obtained by instantiating the place-holders to particular parameters or calls. It is therefore interesting to guide algorithm design by a schema that captures the essence of some strategy.

In order to be more precise, we have to settle for a specific algorithm language. Our focus goes of course to logic algorithms. Informally, in a first approximation, a *logic algorithm schema* is a second-order logic algorithm. A particular logic algorithm, called an *instance* of the schema, is then obtained by instantiating the variables of the schema.

Example 8-1: The following is a logic algorithm schema for the generate-and-test strategy:

```
R(X,Y) ⇔
        Generate(X,Y) ∧ Test(Y)
```

The following logic algorithm for the *sort*/3 predicate is also known as Naive-Sort:

```
sort(L,S) ⇔
        permutation(L,S) ∧ ordered(S)
```

where *ordered(S)* iff *S* is an ascendingly ordered list of integers.This logic algorithm is an instance of the generate-and-test schema above, namely via the second-order substitution {*R/sort, Generate/permutation, Test/ordered, X/L, Y/S*}. ♦

Reality is more complex, however. Function variables and predicate variables may have any arity, and this calls for schema variables to denote these arities. Conjunctions, disjunctions, or quantifications of any length may appear, and this calls for schema variables to denote the ranges of such ellipses. Permutations of parameters, conjuncts, disjuncts, or quantifications may have to be performed in order to see why a logic algorithm is an instance of some schema. Unfold transformations may have to be performed in order to see why a logic algorithm is an instance of some schema.

Example 8-2: Given the logic algorithm schema:

$$R(X_1,\ldots,X_n,Y) \Leftrightarrow P(Y,Z_1,\ldots,Z_n) \wedge \bigwedge_{1\le i\le n} Q_i(X_i,Z_i)$$

it is not immediately clear why the logic algorithm *LA(foo)*:

```
foo(S,B,A) ⇔
    permutation(A,SA)
  ∧ reverse(B,RB)
  ∧ append(SA,RB,S)
  ∧ ordered(SA)
```

is an instance of this schema. Indeed, one of the possible schema substitutions is {*R*/*foo*, *n*/2, X_1/*B*, X_2/*A*, *Y*/*S*, *P*/*append*, Z_1/*RB*, Z_2/*SA*, Q_1/*reverse*, Q_2/*sort*}. But the *sort*/2 atom must also be unfolded into the conjunction of the *permutation*/2 and *ordered*/1 atoms (as in Example 8-1), and a series of permutations of parameters and conjuncts are required to obtain this instance. ♦

A second-order wff schema language is thus needed to write realistic logic algorithm schemas. The formal definition of such a language and its semantics is beyond the scope of this thesis, so we do not develop it. But we know from experience that the intuitive understanding of our schemata is sufficient.

Definition 8-1: A *logic algorithm schema* is a closed second-order wff schema of the form:

$$\forall R\ \forall X_1 \ldots \forall X_n \quad R(X_1,\ldots,X_n) \Leftrightarrow F$$

where n is a schema variable or a constant, the X_i are distinct variables, R is a predicate variable, and F is a second-order wff schema. The atom schema $R(X_1,\ldots,X_n)$ is called the *head*, and F is called the *body* of the logic algorithm schema.

In the sequel, we drop the universal quantifications in front of the heads, as well as any existential quantifications at the beginning of the bodies of logic algorithm schemas.

It is evident that a logic algorithm schema without function variables, predicate variables, and schema variables is a logic algorithm.

Definition 8-2: An *instance* of a logic algorithm schema is a logic algorithm obtained by the following sequence of operations:

(1) permutation of parameters, conjuncts, disjuncts, and quantifications of the schema;
(2) application of a second-order schema substitution to the resulting schema, such that all function variables, predicate variables, and schema variables are instantiated to first-order objects;
(3) application of unfold transformations.

This process is called *instantiation* of a schema. The reverse process is called *classification* of the logic algorithm, and yields a schema called a *cover* of that logic algorithm.

The process of classifying several algorithms into a same schema is called *algorithm analysis*. Interestingly, it may even be seen as *schema synthesis*, because it amounts to second-order schema learning. Similarly, the process of instantiating a given schema into several algorithms is called *algorithm synthesis*, but may also be seen as *schema analysis*.

In order to facilitate "visual" classification of logic algorithms, we introduce the following purely syntactic criterion for the writing of logic algorithms.

Definition 8-3: A *canonical representation* of a logic algorithm wrt a covering schema exactly matches the layout of that schema.

Example 8-3: A possible canonical representation of *LA(foo)* wrt the schema of the previous example is:

```
foo(S,B,A) ⇔
    append(SA,RB,S)
  ∧ reverse(B,RB) ∧ ( permutation(A,SA) ∧ ordered(SA) )
```
♦

Note that a canonical representation is not unique, because of the possible permutations of parameters. The permutations of parameters as in the schema can't be imposed at the in-

stance level, because there is no possible control over how problems, and hence predicates, are defined. We can't thus introduce a concept such as *the normalized representation* of a logic algorithm.

Almost all logic algorithms of this thesis are canonical representations wrt some schema in this chapter.

Note that one may distinguish between design schemas and transformation schemas: the former are useful for guiding algorithm design, whereas the latter are useful for guiding algorithm transformation. But as nothing prevents the use of transformation schemas for guiding algorithm design, the boundary between these two types of schemas seems to be a subjective one. Our views on the differences between design and transformation extend to the differences between design schemas and transformation schemas. Good examples of transformation schemas are the generalization schemas of [Deville 90].

8.2 A Divide-and-Conquer Logic Algorithm Schema

As algorithm design strategies are fairly orthogonal, there is apparently little opportunity for re-use between design steps for different strategies. In this thesis, we focus on the divide-and-conquer strategy, for the following reasons (also see [Smith 85]):

- *diversity*: a wide variety of relations can be implemented by such algorithms;
- *efficiency*: the resulting algorithms often have good time/space complexities;
- *simplicity*: the simplicity of this strategy makes it particularly convenient for (semi-) automated algorithm design.

The support of other strategies is discussed in Section 14.2.3.

In essence, the divide-and-conquer design strategy solves a problem by the following sequence of three steps [Cormen *et al*. 90]:

(1) *divide* a problem into sub-problems, unless it can be trivially solved;
(2) *conquer* the sub-problems by solving them recursively;
(3) *combine* the solutions to the sub-problems into a solution to the original problem.

Hence the name of the strategy. In the sequel, we focus on applying this strategy to data-structures, rather than to states of partial computations.

This strategy description is a little rough, and calls for further details. In Section 8.2.1, we proceed by successive refinements to incrementally infer various versions of a divide-and-conquer logic algorithm schema. Then, in Section 8.2.2, we list the integrity constraints that instances of these schemas have to satisfy in order to be divide-and-conquer algorithms. In Section 8.2.3, we justify the choices that have been made during this algorithm analysis. Finally, in Section 8.2.4, we discuss some other issues related to divide-and-conquer schemas.

8.2.1 Divide-and-Conquer Logic Algorithm Analysis

We now incrementally infer six different versions of a divide-and-conquer logic algorithm schema, starting from the logic algorithms of Chapter 5. Each new version covers a larger set of logic algorithms. As a reminder, we still hypothesize that no parameter is a tuple, that is that procedure declarations are "flattened" out.

First Version

Let's first restrict ourselves to binary relations, and present a most basic strategy that already yields solutions to many problems.

A *divide-and-conquer algorithm* for a binary predicate r over parameters X and Y works as follows. Let X be the induction parameter. If X is minimal, then Y is (usually) easily found by directly solving the problem. Otherwise, that is if X is non-minimal, decompose X into a

```
R(X,Y) ⇔
            Minimal(X)      ∧ Solve(X,Y)
  ∨         NonMinimal(X)   ∧ Decompose(X,HX,TX)
                            ∧ R(TX,TY)
                            ∧ Process(HX,HY)
                            ∧ Compose(HY,TY,Y)
```

Logic Algorithm Schema 8-1: Divide-and-conquer (version 1)

vector ***HX*** of heads of *X* and a vector ***TX*** of tails of *X*, the tails being of the same type as *X*, as well as smaller than *X* according to some well-founded relation. The tails ***TX*** recursively yield tails ***TY*** of *Y*. The heads ***HX*** are processed into a vector ***HY*** of heads of *Y*. Finally, *Y* is composed from its heads ***HY*** and tails ***TY***.

For further discussion, we quantify the vectors as follows. There are *h* heads of *X*, and *h'* heads of *Y*, and *t* tails of *X*, hence *t* tails of *Y*. Thus:

$$\#\boldsymbol{HX} = h$$

$$\#\boldsymbol{HY} = h'$$

$$\#\boldsymbol{TX} = \#\boldsymbol{TY} = t$$

Note that *h*, *h'*, and *t* are schema variables, not constants.

Logic algorithms designed by this basic divide-and-conquer strategy are covered by Schema 8-1, where ***R(TX,TY)*** stands for $\wedge_{1 \le j \le t}\ R(TX_j,TY_j)$, where *j* is a notation variable.

Note that here, and in the sequel, we prefer the verb "decompose" to "divide", and that the "combine" operation is actually split into a "process" operation and a "compose" operation.

Example 8-4: Logic algorithms that are covered by Schema 8-1 are *LA(compress-ext-L)* (LA 5-1), *LA(compress-int-C)* (LA 5-2), *LA(compress-ext-C)* (LA 5-3), *LA(permutation-L)* (LA 5-14), *LA(sort-int-L) {Insertion-Sort}* (LA 5-16), *LA(sort-ext-L) {Quick-Sort}* (LA 5-17), and *LA(sort-ext-L) {Merge-Sort}* (LA 5-18).

Second Version

But many logic algorithms are not covered by Schema 8-1 because the non-minimal case is further partitioned into sub-cases, each featuring a different way of combining partial solutions. The enhanced strategy is as follows.

A *divide-and-conquer algorithm* for a binary predicate *r* over parameters *X* and *Y* works as follows. Let *X* be the induction parameter. If *X* is minimal, then *Y* is (usually) easily found by directly solving the problem. Otherwise, that is if *X* is non-minimal, decompose *X* into a vector ***HX*** of heads of *X* and a vector ***TX*** of tails of *X*, the tails being of the same type as *X*, as well as smaller than *X* according to some well-founded relation. The tails ***TX*** recursively yield tails ***TY*** of *Y*. The heads ***HX*** are processed into a vector ***HY*** of heads of *Y*. Finally, *Y* is composed from its heads ***HY*** and tails ***TY***. It may happen that sub-cases emerge with different processing and composition operators: discriminate between them according to the values of ***HX***, ***TX***, and *Y*.

Of course, if non-determinism of the problem requires alternative solutions, then discriminants should evaluate to *true*. Logic algorithms designed by this enhanced divide-and-conquer strategy are covered by Schema 8-2.

A schema variable *c* represents the number of different sub-cases of the non-minimal case. This new schema supersedes the previous schema if *c* is bound to 1, and the $Discriminate_1$ predicate variable is bound to a predicate that always holds.

Note that the disjunction over *k* could be "pushed" further into the body of the schema: we however prefer the given layout as it preserves the resemblance to the previous version.

```
R(X,Y) ⇔
                Minimal(X)        ∧ Solve(X,Y)
  ∨ ∨_{1≤k≤c}   NonMinimal(X)     ∧ Decompose(X,HX,TX)
                                  ∧ Discriminate_k(HX,TX,Y)
                                  ∧ R(TX,TY)
                                  ∧ Process_k(HX,HY)
                                  ∧ Compose_k(HY,TY,Y)
```

Logic Algorithm Schema 8-2: Divide-and-conquer (version 2)

```
R(X,Y) ⇔
                Minimal(X)        ∧ Solve(X,Y)
  ∨ ∨_{1≤k≤c}   NonMinimal(X)     ∧ Decompose(X,HX,TX)
                                  ∧ Discriminate_k(HX,TX,Y)
                                  ∧ (     SolveNonMin_k(HX,TX,Y)
                                      |
                                        R(TX,TY)
                                    ∧ Process_k(HX,HY)
                                    ∧ Compose_k(HY,TY,Y)      )
```

Logic Algorithm Schema 8-3: Divide-and-conquer (version 3)

Example 8-5: A logic algorithm that is covered by Schema 8-2, but not by the previous schema, is *LA(delOddElems-L)* (LA 5-4).

Third Version

But many logic algorithms are not even covered by Schema 8-2 because the non-minimal case is partitioned into a recursive and a non-recursive case, each of which is in turn partitioned into sub-cases, as in the second version. In the non-recursive case, Y is (usually) easily found by directly solving the problem, taking advantage of the decomposition of X into $\boldsymbol{HX}$ and $\boldsymbol{TX}$. We assume there are v non-recursive sub-cases and w recursive sub-cases, such that $v+w=c$, where c, v, and w are schema variables. Logic algorithms designed by this enhanced divide-and-conquer strategy are covered by the following schema:

```
R(X,Y) ⇔
                Minimal(X)        ∧ Solve(X,Y)
  ∨ ∨_{1≤k≤v}   NonMinimal(X)     ∧ Decompose(X,HX,TX)
                                  ∧ Discriminate_k(HX,TX,Y)
                                  ∧ SolveNonMin_k(HX,TX,Y)
  ∨ ∨_{c-w<k≤c} NonMinimal(X)     ∧ Decompose(X,HX,TX)
                                  ∧ Discriminate_k(HX,TX,Y)
                                  ∧ R(TX,TY)
                                  ∧ Process_k(HX,HY)
                                  ∧ Compose_k(HY,TY,Y)
```

But this schema is very lengthy, and doesn't sufficiently show the commonalities between the recursive and the non-recursive sub-cases. We thus syntactically merge these cases by separating their differences by a BNF-style "or" operator, denoted " | ", which has nothing to do with the logical "or" connective. It is interesting to note that this operator is actually some kind of a second-order "exclusive-or" connective. The result is Schema 8-3.

Example 8-6: A logic algorithm that is covered by Schema 8-3, but not by the previous two schemas, is *LA(member-L)* (LA 5-10).

```
R(X,Y) ⇔
                Minimal(X)        ∧ Solve(X,Y)
  ∨ ∨1≤k≤c      NonMinimal(X)     ∧ Decompose(X,HX,TX)
                                  ∧ Discriminate_k(HX,TX,Y)
                                  ∧ (     SolveNonMin_k(HX,TX,Y)
                                      |
                                          R(TX,TY)
                                    ∧ Process_k(HX,HY)
                                    ∧ Compose_k(HY,TY,Y)       )
```

Logic Algorithm Schema 8-4: Divide-and-conquer (version 4)

Fourth Version

Let's now relax the requirement that predicate *r* be binary. But we keep the (so far implicit) constraint that the induction parameter be simple. Supposing predicate *r* is *n*-ary (where *n* is a schema variable), this new setting implies that *Y* becomes a vector ***Y*** of *n*–1 variables Y_j, and that vector ***TY*** becomes a vector ***$\underline{TY}$*** of *n*–1 vectors $\boldsymbol{TY}_j$, each of which is a vector of *t* variables TY_{jl} (where *j, l* are notation variables). Similarly, ***HY*** becomes a vector ***$\underline{HY}$*** of *n*–1 vectors $\boldsymbol{HY}_j$, each of which is a vector of *h'(j)* variables HY_{jl}, where *h'*/1 is a schema function variable. Thus:

$$\#\boldsymbol{HX} = h$$

$$\#\boldsymbol{HY}_j = h'(j) \quad (1 \le j \le n-1)$$

$$\#\boldsymbol{TX} = \#\boldsymbol{TY}_j = t \quad (1 \le j \le n-1)$$

$$\#\boldsymbol{Y} = \#\underline{\boldsymbol{HY}} = \#\underline{\boldsymbol{TY}} = n-1$$

Logic algorithms designed by this enhanced divide-and-conquer strategy are covered by Schema 8-4, where ***R(TX,$\underline{TY}$)*** stands for $\wedge_{1 \le l \le t}\ R(TX_l, TY_{1l}, \ldots, TY_{n-1l})$.

Example 8-7: Logic algorithms that are covered by Schema 8-4, but not by the previous three schemas, are *LA(efface-L)* (LA 5-5), *LA(firstN-L)* (LA 5-6), *LA(firstN-N)* (LA 5-7), *LA(insert-L)* (LA 5-8), *LA(insert-R)* (LA 5-9), *LA(partition-L)* (LA 5-13), and *LA(plateau-N)* (LA 5-15).

Fifth Version

Let's now relax the requirement that there be exactly one minimal case and exactly one non-minimal case. We assume there are *p* minimal cases, and *q* non-minimal cases, where *p, q* are schema variables. There are *c(i)* sub-cases for non-minimal case *i*, where *c*/1 is a schema function variable. Similarly, there are *h(i)* heads of *X*, and *h'(i,j)* heads of Y_j, and *t(i)* tails of *X*, in non-minimal case *i*, where *h*/1, *h'*/2, and *t*/1 are schema function variables. Thus:

$$\#\boldsymbol{HX}_i = h(i) \quad (1 \le i \le q)$$

$$\#\boldsymbol{HY}_{ij} = h'(i,j) \quad (1 \le i \le q)\ (1 \le j \le n-1)$$

$$\#\boldsymbol{TX}_i = \#\boldsymbol{TY}_{ij} = t(i) \quad (1 \le i \le q)\ (1 \le j \le n-1)$$

$$\#\boldsymbol{Y} = \#\underline{\boldsymbol{HY}}_i = \#\underline{\boldsymbol{TY}}_i = n\text{-}1 \quad (1 \le i \le q)$$

Logic algorithms designed by this enhanced divide-and-conquer strategy are covered by Schema 8-5.

Example 8-8: A logic algorithm that is covered by Schema 8-5, but by none of the previous four schemas, is *LA(parity-L)* (LA 5-12).

```
R(X,Y) ⇔
     ∨1≤i≤p   Minimal_i(X)      ∧ Solve_i(X,Y)
   ∨ ∨1≤i≤q
     ∨1≤k≤c(i) NonMinimal_i(X)  ∧ Decompose_i(X,HX_i,TX_i)
                                ∧ Discriminate_ik(HX_i,TX_i,Y)
                                ∧ (     SolveNonMin_ik(HX_i,TX_i,Y)
                                    |
                                    R(TX_i,TY_i)
                                  ∧ Process_ik(HX_i,HY_i)
                                  ∧ Compose_ik(HY_i,TY_i,Y)    )
```

Logic Algorithm Schema 8-5: Divide-and-conquer (version 5)

```
R(X,Y) ⇔
     ∨1≤i≤p   Minimal_i(X)      ∧ Solve_i(X,Y)
   ∨ ∨1≤i≤q
     ∨1≤k≤c(i) NonMinimal_i(X)  ∧ Decompose_i(X,HX_i,TX_i)
                                ∧ Discriminate_ik(HX_i,TX_i,Y)
                                ∧ (     SolveNonMin_ik(HX_i,TX_i,Y)
                                    |
                                    R(TX_i,TY_i)
                                  ∧ Process_ik(HX_i,HY_i)
                                  ∧ Compose_ik(HY_i,TY_i,Y)    )
```

Logic Algorithm Schema 8-6: Divide-and-conquer (version 6)

Sixth Version

Let's finally relax the constraint that the induction parameter be simple. We assume the induction parameter is composed of x parameters, while there are y other parameters, where x, y are schema variables. This implies that $\boldsymbol{HX}_i$ becomes a vector $\underline{\boldsymbol{HX}}_i$ of x vectors $\boldsymbol{HX}_{ij}$, each of which is a vector of $h(i)$ variables HX_{ijl}. Similarly, $\boldsymbol{TX}_i$ becomes a vector $\underline{\boldsymbol{TX}}_i$ of x vectors $\boldsymbol{TX}_{ij}$, each of which is a vector of $t(i)$ variables TX_{ijl}. Hence:

$$\#\boldsymbol{X} = \#\underline{\boldsymbol{HX}}_i = \#\underline{\boldsymbol{TX}}_i = x \quad (1{\le}i{\le}q)$$

$$\#\boldsymbol{HX}_{ij} = h(i) \quad (1{\le}i{\le}q)\ (1{\le}j{\le}x)$$

$$\#\boldsymbol{HY}_{ij} = h'(i,j) \quad (1{\le}i{\le}q)\ (1{\le}j{\le}y)$$

$$\#\boldsymbol{TX}_{ij} = t(i) \quad (1{\le}i{\le}q)\ (1{\le}j{\le}x)$$

$$\#\boldsymbol{TY}_{ij} = t(i) \quad (1{\le}i{\le}q)\ (1{\le}j{\le}y)$$

$$\#\boldsymbol{Y} = \#\underline{\boldsymbol{HY}}_i = \#\underline{\boldsymbol{TY}}_i = y \quad (1{\le}i{\le}q)$$

Logic algorithms designed by this enhanced divide-and-conquer strategy are covered by Schema 8-6.

Example 8-9: A logic algorithm that is covered by Schema 8-6, but by none of the previous five schemas, is *LA(merge-<A,B>)* (LA 5-11).

Things are already getting very complicated with the fifth version. But *LA(split)* (LA 5-19) is still uncovered, which means that the search for even more general versions could continue. But we stop it here. Some of the possible extensions are outlined in Section 8.4. The remainder of this discussion is mostly about version 4.

8.2.2 Integrity Constraints on Instances

The schemas above can be instantiated in many ways. However, some constraints need to be verified by these instantiation processes in order to result in valid divide-and-conquer logic algorithms. The constraints on instances of version 4 are as follows.

The instance of X, that is the induction parameter, must be of an inductive type. Indeed, otherwise, its decomposition into tails $\boldsymbol{TX}$ that are each smaller than X according to some well-founded relation would be impossible.

The minimal form and the non-minimal form must be mutually exclusive over the domain of the induction parameter. This means that the chosen instance of the formula:

$$X \in dom(\mathcal{R}) \Rightarrow Minimal(X) \veebar NonMinimal(X) \tag{1}$$

must be *true* in $\Im$, where $\veebar$ denotes the "exclusive or" connective. This constraint considerably facilitates algorithm design. Of course, as seen earlier, a rewriting of the final logic algorithm may blur this distinction, which is thus only mandatory at the level of canonical representations of divide-and-conquer logic algorithms.

The instance of *NonMinimal* must be a precondition for the instance of *Decompose*. And the instance of *Decompose* must be deterministic if X is given. This means that the chosen instance of the formula:

$$X \in dom(\mathcal{R}) \wedge NonMinimal(X) \Rightarrow \exists!\boldsymbol{HX}\, \exists!\boldsymbol{TX}\ \ Decompose(X,\boldsymbol{HX},\boldsymbol{TX}) \tag{2}$$

must be *true* in $\Im$. This is a reasonable constraint, as it facilitates algorithm design, especially in conjunction with constraint (1). The determinism of *Decompose* is not necessary for correctness reasons, but is reasonable in view of efficiency of the synthesized algorithm.

The decomposition of X must yield tails TX_i that are each smaller than X according to some well-founded relation "<". This means that the following formula:

$$\exists \text{"}<\text{"}\ \ \forall (Decompose(X,\boldsymbol{HX},TX_1,\ldots,TX_t) \Rightarrow \forall i \in \{1,\ldots,t\}\ \ TX_i \text{ "}<\text{" } X) \tag{3}$$

must be *true* in $\Im$. This ensures termination of the algorithm in the "all ground" mode. Note that a similar constraint need not be imposed on the $\boldsymbol{TY}$.

The constraints on the instances of the schema variables (all versions) are as follows:

$$n \geq 1 \tag{4}$$

$$x \geq 1 \ \wedge\ y \geq 0 \ \wedge\ x+y = n \tag{5}$$

$$p \geq 1 \ \wedge\ q \geq 1 \tag{6}$$

$$c \geq 1, \text{ respectively } c(i) \geq 1 \quad (1 \leq i \leq q) \tag{7}$$

$$h \geq 0, \text{ respectively } h(i) \geq 0 \quad (1 \leq i \leq q) \tag{8}$$

$$h' \geq 0, \text{ respectively } h'(j) \geq 0, \text{ respectively } h'(i,j) \geq 0 \quad (1 \leq i \leq q)\ (1 \leq j \leq y) \tag{9}$$

$$t \geq 1, \text{ respectively } t(i) \geq 1 \quad (1 \leq i \leq q) \tag{10}$$

Constraints (4) and (5) state that 0-ary relations cannot be solved by a divide-and-conquer approach, and that there must be some induction parameter. Constraint (6) requires that there must be at least one minimal case, and at least one non-minimal case. Constraint (7) says that each non-minimal case must have at least one sub-case. Constraints (8) to (10) state that every parameter must be decomposable into at least one tail and zero heads.

Note that there is no constraint that the primitive =/2 must be used in the definition of the instances of *Minimal* and *NonMinimal*. We did so in Section 5.2, but this is not necessary. For instance, if the minimal form is [], and the non-minimal form is [_|_], the following instances of *NonMinimal* could be used:

$$nonMinimal(L) \Leftrightarrow L=[\_|\_]$$

$$nonMinimal(L) \Leftrightarrow L\neq[]$$

$$nonMinimal(L) \Leftrightarrow length(L,N) \wedge N>0$$

Up to domain checking, these variants are equivalent.

It is also worth noting that the ***HX*** (respectively ***HY***) may be of the same type as *X* (respectively *Y*). Indeed, there is no reason why, say, *HL* should be an integer when *L* is a list of integers. This is illustrated by *LA(compress-ext-L)* (LA 5-1), where *HL* is a list of integers.

8.2.3 Justifications

A series of deliberate decisions have been made during the algorithm analysis above. Let's justify them now.

Instances of *Solve* may be defined by fairly complex formulas, including divisions into sub-cases and the corresponding discriminating mechanisms, just as in non-minimal cases. But since this is relatively exceptional, we prefer to keep the schema simple, and always stuff such formulas into a single predicate variable.

Such is however not the case with the $SolveNonMin_k$, where *Decompose* and $Discriminate_k$ are explicitly present: this is for reasons of homogeneity with the recursive non-minimal case. This even implies that instances of the $SolveNonMin_k$ may use the variables ***HX*** and ***TX*** introduced by *Decompose*, rather than start from scratch with a non-decomposed *X*.

Instances of *Solve* and the $SolveNonMin_k$ are fundamentally different in nature from instances of the $Process_k \wedge Compose_k$ conjunctions: in the former, *Y* can be anything, even totally unrelated to *X*, ***HX***, or ***TX***; in the latter, *Y* must be in terms of ***TY*** at least.

The distinction between *NonMinimal* and *Decompose* may seem artificial at first sight. Indeed, in many of the logic algorithms of Chapter 5, their instances can be unified, such as in *LA(firstN-L)* (LA 5-6). But the mission of *NonMinimal* only is to detect a non-minimal form, whereas the mission of *Decompose* is to decompose a form that is known to be non-minimal: these are two totally different missions, and the integrity constraint above on the relationship between their instances indeed is an implication, not an equivalence. This is also reflected in the different parameters to the corresponding predicate variables. Sample logic algorithms that clearly illustrate these differences are *LA(compress-ext-L)* (LA 5-1), *LA(sort-ext-L){Quick-Sort}* (LA 5-17), and *LA(efface-L)* (LA 5-5).

In the first four versions (where there is only one minimal case and only one non-minimal case), *NonMinimal(X)* can be rewritten as $\neg$*Minimal(X)* iff this preserves constraint (2). But, in view of preserving the similarity with the last two versions, we prefer to make the distinction explicit right away. Also, and more importantly, there is no need for such an arbitrary restriction on possible instances of *NonMinimal*.

8.2.4 Discussion

Some issues about the divide-and-conquer strategy need to be discussed in order to show its generality, and clearly distinguish it from some other approaches.

First, the divide-and-conquer strategy is often believed to be restricted to algorithms that involve some sophisticated design decisions. A famous example is the Quick-Sort algorithm, where the divide step partitions the given list into two sublists of elements that are greater (respectively smaller) than a pivot element. However, such design decisions only affect the complexity of the resulting algorithm, and are thus not strictly necessary for the design of correct algorithms. Hence, a divide step that simply decomposes a list into its head and its tail is also valid. In the sorting problem, it leads to Insertion-Sort.

Also, we mentioned that step (1) of a divide-and-conquer strategy consists of "dividing a problem into sub-problems, unless it can be trivially solved". We have here taken the option that the "unless it can be trivially solved" clause is applicable iff a minimal form of the do-

main of the induction parameter is attained. An alternative interpretation would be that the clause may be applicable in even other cases. A good illustration of this point of view is Sedgewick's enhancement of Hoare's original Quick-Sort algorithm: it switches to Insertion-Sort once the unsorted list has less than, say, 15 elements. Such eminently sophisticated design-choices are beyond the scope of our study.

An apparent disadvantage of the divide-and-conquer strategy is that it seems to lead to logic algorithms, and hence logic programs, that are not tail-recursive, because of the very placement of recursion in the schema. This argument can however be disputed because (automatable) transformation techniques exist for obtaining tail-recursive versions of a program, and this in many cases [Deville 90].

The divide-and-conquer strategy should not be confused with the top-down decomposition strategy, despite their almost synonymic names and their overall problem reduction approaches. Indeed, the former is a very precisely defined strategy that always yields recursive algorithms, whereas the latter almost never yields recursive algorithms. Note however that the divide-and-conquer strategy may be applied recursively, yielding a top-down approach. This amounts to re-applying the divide-and-conquer strategy in order to synthesize the instances of some predicate variables of the divide-and-conquer schema.

Finally, what is the relationship between the divide-and-conquer strategy and the methodology of algorithm construction by structural induction, as described in Chapter 4? Fundamentally, there is none: both strategies are based on the principle of well-founded induction. However, in practice, there are some differences. The divide-and-conquer strategy, especially its formalization as a schema, is much more prescriptive and more precisely defined than the other methodology. For instance, the first-order variables of a divide-and-conquer schema have well-defined scopes. A good illustration of this phenomenon is that the logic algorithms of Chapter 4 are designed by the structural induction methodology, but are not obtainable by simply instantiating some divide-and-conquer schema. Such schema instances usually look a little bit contrived compared to their more natural-looking hand-constructed counterparts. But it is precisely this extreme formalization that allows (partial) mechanization of the design process. The point thus merely is that both methodologies were formulated with different objectives in mind: hand-construction, respectively automation.

8.3 Stepwise, Schema-Guided Logic Algorithm Synthesis

Let's reconsider the strategies of stepwise synthesis of logic algorithms, seen in Section 7.3. An interesting idea, especially with non-incremental stepwise strategies, is then to establish a mapping between steps and the variables of a schema: each step synthesizes instance(s) of some predicate or schema variable(s) of a given schema.

Example 8-10: Given Schema 8-3, a possible stepwise strategy would be the following fixed sequence of steps:

- Step 1: Syntactic creation of a first approximation;
- Step 2: Synthesis of *Minimal* and *NonMinimal*;
- Step 3: Synthesis of *Decompose*;
- Step 4: Syntactic introduction of the recursive atoms;
- Step 5: Synthesis of *Solve* and the $SolveNonMin_k$;
- Step 6: Synthesis of the $Process_k$ and $Compose_k$;
- Step 7: Synthesis of the $Discriminate_k$.

This sequence of predicate variable instantiations is quite "natural". It can be justified as follows. Due to interdepencies within the schema, there is no discussion as to the mission of Steps 1, 2, and 7. Also, Steps 5 and 6 have missions that can be accomplished in parallel.

Finally, there is no prejudice about what happens first at Step 6: the predicates could be synthesized serially or in parallel. But an alternative sequence nevertheless emerges: one might first choose an adequate composition operator for parameter *Y*, and then reason backwards in order to infer the corresponding decomposition operator for parameter *X*. As a result, Step 3 would be interchanged with Steps 5 and 6. A similar analysis is made by [Smith 85]. However, due to the multi-directionality (or reversibility) of logic algorithms, we can here claim that the given sequence and its outlined alternative are isomorphic: choosing a composition operator *compose* for parameter *Y* amounts to actually selecting *Y* as the induction parameter and using *compose*, in its reversed directionality, as a decomposition operator for *Y*. In other words, the given sequence is probably the only one. ♦

One of the major ideas of this thesis is that schema-independent methods can be developed for the synthesis of predicate variables. Such methods may be merely based on databases of useful instances of predicate variables. More sophisticated methods would perform actual computations for inferring such instances. Possible modes of reasoning are inductive inference, deductive inference, analogical inference, abductive inference, and so on. These reasonings would be based on the contents of the specifications, as well as the algorithm as designed so far. Several methods of such a tool-box might be applicable at each step, thus yielding opportunities for user interaction, or for the application of heuristics.

We thus advocate a more disciplined approach to algorithm synthesis: rather than use a uniform method for instantiating all variables of a given schema (possibly without any awareness of such a schema), one should deploy for each variable the best-suited method. In other words, we propose to view research on automatic programming as (1) the search for adequate schemas, (2) the development of useful methods of predicate variable instantiation, and (3) the discovery of interesting mappings between these methods and the variables of these schemas.

As many methods would be schema-independent, and hence variable-independent, one could even investigate synthesis methodologies that are parameterized on schemas. In other words, a Step 0 would be to select an appropriate schema, and the subsequent steps would be either a hardwired sequence (specific to the selected schema) of applications of methods, or a user-guided selection of variables and methods. Our grand view of algorithm synthesis systems thus is one of a large workbench with a disparate set of highly specialized methods for a set of schemas that covers (as much as possible of) the space of all possible algorithms.

Note that this discussion is independent of the used specification formalism, and hence of their completeness or incompleteness. In this thesis, we investigate a particular niche of automatic algorithm synthesis as defined above: in the context of (incomplete) specifications by examples and properties, develop methods of predicate variable instantiation, and apply them to the step/variable mapping identified in Example 8-10.

8.4 Future Work

There are several directions for future research.

First, in view of having a *theory of logic algorithm schemas* and a formalization of the accompanying notions of instantiation and classification, a language for second-order wff schemas needs to be developed, as well as its semantics defined.

Second, as we already have hinted when stopping the incremental inference of different versions of a divide-and-conquer schema, version 6 of that schema is far from covering all possible divide-and-conquer logic algorithms.

A possible enhancement is a schema that automatically introduces an additional parameter to the specified predicate, hence covering *LA(split)* (LA 5-19). Such an extension is dis-

cussed by [Summers 77]. This schema could still be considered a design schema, rather than a transformation schema, as it is not possible to design a logic algorithm for *split*/3 that is covered by one of the given versions.

Another observation is that no operator is applied to the final value of parameter *Y*. For instance, problems like:

average(L,A) iff *A* is the average value of *L*,
where *A* is an integer, and *L* is a non-empty integer-list.

are beyond the scope of the given schemas, and must be solved via a different approach. For instance, it could be solved by a top-down decomposition:

```
average(L,A) ⇔
                sum(L,S) ∧ length(L,N) ∧ div(S,N,A)
```

and possible optimization by loop-merging the logic algorithms for *sum*/2 and *length*/2. Another solution would be to extend the divide-and-conquer schemas accordingly.

Also, in Chapter 5, we have informally introduced the notion of *auxiliary parameter*. But in this chapter, we have completely ignored them. This is justifiable by the observation that the identification of auxiliary parameters is not necessary at all for correct algorithm design. Indeed, as the logic algorithms of Section 5.2 show, it is possible to write logic algorithms that don't distinguish between auxiliary parameters and "ordinary" parameters. However, the (de)composition of an auxiliary parameter from (into) its heads and tails may look cumbersome because an auxiliary parameter *Y* and its tail *TY* are eventually found to be identical: *Y=TY*. But it is precisely this composition pattern that allows the subsequent detection of auxiliary parameters, and their elimination from the decomposition machinery, so as transform the logic algorithm into a more "graceful" and "natural" version. For instance, *LA(efface-L)* (LA 5-5) could be rewritten as follows:

```
efface(E,L,R) ⇔
    L=[_]      ∧ L=[HL] ∧ E=HL ∧ R=[]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
                ∧ HL=E
                ∧ E=HL ∧ R=TL
  ∨ L=[_,_|_] ∧ L=[HL|TL]
                ∧ HL≠E
                ∧ efface(E,TL,TR)
                ∧ HR=HL
                ∧ R=[HR|TR]
```

We should not forget that just a casual glance at a specification will not tell whether a parameter is an "ordinary" or an auxiliary one. Things are even more difficult with automated algorithm design, and the surest way is indeed to ignore the potential existence of auxiliary parameters until a transformation phase. But suppose now that knowledge about which parameters are auxiliary parameters is available earlier during the algorithm synthesis process (for instance because the specifier declares them as such, or because type heuristics detect them as such). It would certainly be helpful to pre-compile the needed subsequent transformations into a schema with an explicit consideration of auxiliary parameters.

Third, this chapter is exclusively concerned with the divide-and-conquer strategy. It would be interesting to investigate the development of schemas for other strategies, such as generate-and-test, local search, global search, and so on.

8.5 Related Work

Algorithm (and program) schemas are an old, and ever popular, idea of computer science. They have been proposed for a huge variety of applications.

Some *properties of programs* can be proven independently of their actual computations. Hence the idea of abstracting away these details, and proving such properties for the resulting schemas. This makes proofs of properties at the instance level easier, as it suffices to show that the program is covered by a schema that is known to have the desired property. Sample properties are termination, divergence, equivalence, isomorphism, and so on. An early survey of this research can be found in [Manna 74]. Note that Manna's schemas are first-order schemas, and that their semantics is defined via (Herbrand) interpretations. This semantics-based approach is of course also perfectly valid. It is sufficient for the study of schemas, but not as a tool for algorithm design. Indeed, unlike our second-order approach, it doesn't permit the concept of instantiation of a schema, and is thus less "constructive" for design purposes.

In the field of *logic programming tutors* for beginners, [Gegg-Harrison 89, 93] proposes a hierarchy of 14 logic program schemas. These are also set in a second-order logic framework, and embody the otherwise rare feature of arbitrary arities. They are specialized versions of our divide-and-conquer schemas, in the sense that they have less predicate variables, and that they are already partly instantiated (induction parameter of type list, fixed forms, fixed decomposition, fixed number of cases, fixed discriminants, and so on).

In the area of manual or computer-aided *algorithm/program construction* for experts, [Deville 87, 90] and [Deville and Burnay 89] suggest an ancestor version of our divide-and-conquer schemas. It roughly corresponds to a highly instantiated version of our version 1 (no discriminants, and most schema variables are bound to the constant 1). They also discuss interesting transformation schemas, based on structural or computational generalization.

A similar study is made by [O'Keefe 90], who rephrases "specifications" of problems in an algebraic way. The predicates of such "specifications" can be directly plugged into given logic program schemas. Several schemas may be applicable according to the properties (associativity, commutativity, existence of left identities, and so on) of the identified predicates.

Alternatively, [Lakhotia 89] experiments with what he calls "incorporating programming techniques into Prolog programs". Similarly, [Barker-Plummer 90] discusses a system based on clichés that assists experienced programmers in the construction of Prolog programs. These approaches could be seen as transformation rather than design approaches.

The field of automatic *algorithm/program synthesis* has naturally seen a lot of interest in schemas. The promise of schema-guidance is a more disciplined synthesis that exploits useful knowledge about algorithm design strategies.

In the sub-area of synthesis from incomplete specifications (see Section 3.3), schemas are often implicitly or explicitly present to guide the synthesis. Early systems based on divide-and-conquer schemas are described by [Shaw *et al.* 75], [Hardy 75], [Summers 77], and [Biermann and Smith 79].

Surprisingly, the new strand of research spawned by MIS [Shapiro 82] (see Section 3.4.1) has long ignored the virtues of schemas. It is only in these days that schemas have been found to be interesting for organizing search spaces for MIS-like systems. Sample approaches are described by [Tinkham 90] and [Sterling and Kirschenbaum 91].

In the sub-area of synthesis from complete specifications, the use of schemas seems much less established. The *Programmer's Apprentice* project is based on the fundamental notion of clichés, which are application-specific schemas [Rich and Waters 88b].

Some of the most fascinating research on using schemas for program synthesis is being reported by Douglas R. Smith. The synthesis of divide-and-conquer algorithms is described by [Smith 81, 85]: the underlying schema is quite similar to our version 4, except that the

discriminants are merged into the form-identifying formulas. With the implemented system, called CYPRESS, one proceeds by arbitrarily choosing instances of two predicate variables among <*Decompose, Process, Compose*>, and then infers the other ones deductively using a synthesis theorem.

The synthesis of global search algorithms is reported by [Smith 88, 90]. Global search is an enumerative search strategy that generalizes many known search strategies, such as binary search, backtracking, branch-and-bound, constraint satisfaction, and so on. The resulting system, called KIDS, is similar in nature to the CYPRESS approach.

As a conclusion of this survey, it must be observed that almost all of the research has gone into investigating divide-and-conquer schemas. This is not a bad thing, as the covered class of algorithms is very interesting and large. But these published schemas tend to be extremely simplified ones, namely often at best the equivalents of our version 2. In order to be really useful, we think that more sophisticated versions should also be supported.

8.6 Conclusion

In this chapter, we have given a brief semi-formal introduction to algorithm schemas, and argued for their usefulness in guiding algorithm design. Focusing on the divide-and-conquer strategy, we have incrementally inferred six increasingly powerful schemas from a series of divide-and-conquer logic algorithms. These schemas are data-structure independent. Finally, we have proposed a vision of stepwise, schema-guided algorithm synthesis mechanisms, where each variable of a schema is instantiated using the best-suited method from a tool-box of such methods.

Algorithm schemas are used in a widespread variety of areas, including automated algorithm synthesis. This is quite natural, as algorithm schemas are a powerful way of embodying our knowledge about algorithms. Surprisingly, though, schemas are not as broadly used across all existing synthesis mechanisms as we think they ought to be. For instance, in the branch of inductive synthesis, schemas had almost disappeared until recently. One of the reasons could be that Inductive Logic Programming (ILP) seems more focused on concept learning than on algorithm design, and it is little wonder that concept schemas are hardly existent, and thus considered only with skepticism. Similarly, in the branch of deductive synthesis, schemas have attracted little attention. We think that deductive synthesis paradigms as proofs-as-programs synthesis, or transformational synthesis, would gain a lot from schema-guided approaches, as search spaces can then be significantly cut down.

9 The Proofs-as-Programs Method

In this chapter, we develop the Proofs-as-Programs Method, which adds atoms to a logic algorithm so that some correctness criteria wrt a set of properties become satisfied. This method is part of our tool-box of methods for instantiating the predicate variables of a schema. First, in Section 9.1, we state the problem. Then, in Section 9.2, we explain a method to solve this problem, and discuss its correctness in Section 9.3. In Section 9.4, we illustrate this method on a few sample problems. Future work and related work are discussed in Section 9.5 and Section 9.6, respectively, before drawing some conclusions in Section 9.7.

9.1 The Problem

Let's first illustrate the objectives on a sample problem.

Example 9-1: Intuitively, the first objective consists of adding atoms to a logic algorithm so that it becomes totally correct wrt a given set of properties. For instance, given the logic algorithm *LA(efface)*:

```
efface(E,L,R) ⇔
    L=[HL|TL] ∧ R=TL
  ∨ L=[HL|TL] ∧ efface(E,TL,TR)
              ∧ R=[HL|TR]
```

and the set of properties $\mathcal{P}$*(efface)*:

```
efface(X,[X|T],T)
efface(X,[Y,X|T],[Y|T]) ⇐ X≠Y
```

one can see that *LA(efface)* is complete, but not totally correct, wrt $\mathcal{P}$*(efface)*. Indeed, some atoms are missing for deciding when *R* should be unified with *TL* (namely iff *HL=E*), respectively when *R* should be unified with [*HL*|*TR*] (namely iff *HL*≠*E* and the first element of *TL* is *E*). The objective is thus to add the missing literals to *LA(efface)*, yielding *LA'(efface)*:

```
efface(E,L,R) ⇔
    L=[HL|TL] ∧ HL=E ∧ R=TL
  ∨ L=[HL|TL] ∧ HL≠E ∧ TL=[E|_]
              ∧ efface(E,TL,TR)
              ∧ R=[HL|TR]
```

which is indeed totally correct wrt $\mathcal{P}$*(efface)*. However, as properties are usually an incomplete source of information about the intended relation, we are not necessarily interested in achieving such a total correctness wrt the given property set. We are thus rather interested in achieving total correctness wrt the intended relation. The second objective is thus to generalize *LA'(efface)* into *LA"(efface)* so that it becomes "more complete" wrt $\mathcal{E}$*fface*, and this with some reasonable certainty. In our case, this may be achieved by dropping one of the initially added conditions (namely that the first element of *TL* be *E*). So *LA"(efface)* is:

```
efface(E,L,R) ⇔
    L=[HL|TL] ∧ HL=E ∧ R=TL
  ∨ L=[HL|TL] ∧ HL≠E
              ∧ efface(E,TL,TR)
              ∧ R=[HL|TR]
```

which is indeed totally correct wrt $\mathcal{E}$*fface*. In general, the conditions to be dropped are determined by some application-specific heuristics, whereas the initial addition of conditions seems feasible in an application-independent fashion. ♦

Formally, the problem can now be posed as follows.
Given:

- a logic algorithm *LA(r)*,
- a set of properties $\mathcal{P}(r)$,
- a set $\mathcal{H}$ of application-specific generalization heuristics,

such that:

- *LA(r)* only contains primitive predicates, and the *r/n* predicate,
- *LA(r)* is complete wrt $\mathcal{P}(r)$,

find:

- a logic algorithm *LA'(r)*,
- a logic algorithm *LA"(r)*,

such that:

- *LA'(r)* is a syntactic specialization of *LA(r)* that is totally correct wrt $\mathcal{P}(r)$,
- *LA"(r)* is obtained by applying the heuristics of $\mathcal{H}$ to *LA'(r)*.

The latter task cannot be further explained here, as the heuristics are application-specific. A semantic specialization of *LA(r)* would be $\mathcal{P}(r)$ re-expressed as a logic algorithm. The most "useful" specialization is however a syntactic specialization that only adds atoms to the existing disjuncts of *LA(r)*, and hence preserves the existing computations. Indeed, the objective is not that *LA'(r)* be totally correct wrt $\mathcal{P}(r)$, but that *LA"(r)* be totally correct wrt $\mathcal{R}$. So the construction of *LA'(r)* is only a useful intermediate step.

9.2 A Method

First of all, let's remark that our Proofs-as-Programs Method is totally unlike classical proofs-as-programs approaches to algorithm synthesis (see Chapter 2). Indeed, we here adopt the liberal viewpoint that any technique that extracts some computational contents from a proof may be seen as a proofs-as-programs technique.

Intuitively, the Proofs-as-Programs Method specializes the given logic algorithm by adding atoms extracted from the proofs that the given properties are its logical consequences. After deciding on some concrete language issues, this section is organized as follows: Section 9.2.1 is about the proof aspects of the Proofs-as-Programs Method, whereas Section 9.2.2 is about its condition extraction aspects.

The concrete languages for logic algorithms and properties need to be fixed before actually developing a method for solving the problem at hand. In the following, we assume that:

- logic algorithms have bodies in prenex disjunctive normal form, without negation;
- properties are Horn clauses with atoms of predicate *r/n* in their heads.

These choices are motivated by the theorem proving technology used below. Note that the language for properties is different from the language chosen in Section 6.2, because recursion is allowed here, while negation is disallowed here.

9.2.1 Proofs by Extended Execution

The idea is to perform a verification proof that *LA(r)* is effectively complete wrt $\mathcal{P}(r)$. This should reveal, in case of success, a set of conditions explaining why, if at all, *LA(r)* is not already totally correct wrt $\mathcal{P}(r)$.

If we want to perform this proof in a mechanized fashion, we cannot apply the actual criterion of completeness (namely Definition 7-8) of a logic algorithm wrt a property set, because this requires the knowledge of $\Im$. So we have to resort to the (weaker) intuitive criterion of completeness. The latter would be achieved if (but not only if) $LA(r) \models \mathcal{P}(r)$. But we rather want to prove, for each property $P_i \in \mathcal{P}(r)$, that:

$$\mathcal{T}_i \models P_i \tag{1}$$

where $\mathcal{T}_i$ is a theory composed of:

- the logic algorithm *LA(r)*,
- the specification $\mathcal{P}(r) \setminus \{P_i\}$,
- logic algorithms for all primitive predicates.

The reason why the properties other than P_i are included in $\mathcal{T}_i$ is that the recursive calls of *LA(r)* had better not be resolved using *LA(r)* itself, as we have no guarantee yet of the total correctness of *LA(r)* wrt the intended relation $\mathcal{R}$. So these other properties should be used for the resolution of these recursive calls, because properties are assumed to be a consistent, albeit incomplete, approximation of $\mathcal{R}$. This is reflected in the search rule described below.

These proofs have to be performed by an extension to SLD resolution, because the initial goals are here Horn clauses, and not mere conjunctions of atoms. An existing such extension is the *extended execution* mechanism of [Kanamori and Seki 86] [Kanamori and Fujita 86], where theories are definite programs, and goals are so-called *S-formulas* (short for *specification formulas*). Note that most of the terminology (but not the notation) used hereafter is borrowed from these two original references. This passage only states results, but not their motivations or proofs: refer to these references when in need of further explanations.

We syntactically restrict *S*-formulas (which are not explained here) to *implicative goals* (as in [Fribourg 90, 91a]), that is statements G of the form:

$$\forall \boldsymbol{X}\, \exists \boldsymbol{Y} \quad G^+(\boldsymbol{X},\boldsymbol{Y}) \Leftarrow G^-(\boldsymbol{X})$$

where *conclusion* G^+ and *hypothesis* G^- are conjunctions of atoms, and $\boldsymbol{X}$, $\boldsymbol{Y}$ are vectors of (*universal*, respectively *existential*) variables. For syntactic convenience, we write implicative goals in quantifier-free form:

$$G^+(\boldsymbol{X},?\boldsymbol{Y}) \Leftarrow G^-(\boldsymbol{X})$$

where $\boldsymbol{X}$, $?\boldsymbol{Y}$ are (now) vectors of (*free*, respectively *undecided*) variables. In the sequel, we often write "goal" instead of "implicative goal".

Properties are thus a particular case of implicative goals. Note that $?\boldsymbol{Y}$ is necessarily always empty for a property.

As a theory of extended execution is a definite program, a definite program version of $\mathcal{T}_i$ has to be generated: let's call it $\mathcal{D}_i$. In definite program clauses, we use the connectives "←" and "," for "if" and "and", respectively. This is straightforward due to the chosen languages for logic algorithms and properties. This is explained, for instance, in [Deville 90, pages 227–228] and [Lloyd 87, page 113]. Note that such a translation is deterministic in this case, and thus reversible. Objective (1) thus amounts to proving the following:

$$\mathcal{D}_i \vdash P_i \tag{2}$$

The variables of $\mathcal{D}_i$ are renamed at each use so that there is no conflict with the variable names of previous goals.

The *initial goal* is property P_i, translated into an implicative goal.

Due to our syntactic restriction to implicative goals, only three inference rules of extended execution are actually required in our context. [13]

Definition 9-1: The rule of *definite clause inference* (denoted *DCI*) is a natural extension of SLD resolution to implicative goals. Given a goal G, the selected atom is chosen within G^+, and the mgu may only bind undecided variables of G^+. All new variables introduced in the resolvent goal are undecided variables.

13. Note that we could have adapted extended execution to handle our original formalisms without translations. But we feel that re-using (subsets of) existing results (namely [Kanamori and Fujita 86] [Kanamori and Seki 86] for execution, and [Deville 90] for the necessary translations) is a better approach.

Definition 9-2: The rule of *negation-as-failure inference* (denoted *NFI*) is a natural extension of the NF rule [Clark 78] to implicative goals. Given a goal G, the selected atom A is chosen within G^-, and a conjunction of d resolvent goals is generated, namely $G\sigma_i$, where $A\sigma_i$ has been replaced by the conjunction $B_i\sigma_i$, with $H_i \leftarrow B_i$ being one of the d definite clauses whose head H_i unifies with A under mgu σ_i. All new variables introduced in the resolvent goals are free variables.

Definition 9-3: Given a goal G, the rule of *simplification* (denoted *Sim*) selects two atoms A and B, in G^+ and G^- respectively, that unify with an mgu σ that only binds undecided variables of G^+: the resolvent goal is obtained from $G\sigma$ by deleting A and B.

This subset of extended execution is sound and complete wrt the Clark completion semantics [Kanamori 86]. Hence, provability by extended execution is equivalent to truth. Moreover, since $\mathcal{T}_i$ is by construction the completion of $\mathcal{D}_i$, the proofs of (2) thus effectively do amount to the truths of (1).

SLD resolution is parameterized on a computation rule and a search rule [Lloyd 87]. For extended execution, these rules are, for the purpose of the Proofs-as-Programs Method, instantiated as follows:

- the *computation rules* for *DCI* and *NFI* satisfy the following condition: never select an atom with predicate *r/n* if there are atoms with primitive predicates. Indeed, while theoretically not required, delaying the selection of *r/n* atoms generally results in less search;
- the *search rule* for DCI is as follows:
  - an atom with a primitive predicate is resolved according to its semantics;
  - the atom with predicate *r/n* originating from the initial goal of the proof tree is resolved using the clauses generated from *LA(r)*;
  - an atom with predicate *r/n* not originating from the initial goal of the proof tree is resolved using the clauses generated from $\mathcal{P}(r) \setminus \{P_i\}$.

Note that the search rule is context-dependent. For the resolution of the atom with predicate *r/n* originating from the initial goal, we use *LA(r)* rather than $\mathcal{P}(r) \setminus \{P_i\}$, because that wouldn't make sense: we are trying to prove *LA(r)* complete wrt $\mathcal{P}(r)$, not to prove $\mathcal{P}(r)$ internally consistent. For the resolution of atoms with predicate *r/n* not originating from the initial goal, we use $\mathcal{P}(r) \setminus \{P_i\}$ rather than *LA(r)*, because the latter is in general not correct wrt $\mathcal{P}(r)$.

Note that there is independence of the computation rules. But a fairness condition requires that no atom be indefinitely ignored by *NFI*. The completeness proof of extended execution [Kanamori 86] states that every logical consequence is provable by a so-called *normalized derivation*, in which a sequence of *NFI* inferences precedes a sequence of *DCI* inferences, which in turn precedes a sequence of *Sim* inferences. Hence the idea of a priority ordering between the inference rules, namely *NFI* > *DCI* > *Sim*.

Definition 9-4: A derivation via the *NFI*, *DCI*, and *Sim* rules *succeeds* iff it ends in a goal whose conclusion is empty. Such a derivation *fails* iff it doesn't succeed.

Failure is not always detectable, except in specific settings. For instance, no infinite derivation can occur if all primitive predicates have finite proofs for all directionalities, and if there is a well-founded relation between the recursive atoms of a property and the head of that property.

Before proceeding with the theoretical considerations, let's illustrate all this on a sample derivation.

Example 9-2: Given the definite clause theory $\mathcal{D}_2$:

```
firstPlateau(L,P,S) ← L=[_,_|_],
                      L=[HL|TL],
                      firstPlateau(TL,TP,TS),
                      P=[HL|TP],S=TS,TP=[HL|_]      (C3)
firstPlateau([X],[X],[])                            (P1)
firstPlateau([X,Y],[X],[Y]) ← X≠Y                   (P3)
```

and the implicative goal:

```
firstPlateau([X,Y],[X,Y],[]) ⇐ X=Y                  (P2)
```

here follows the proof that $\mathcal{D}_2 \vdash P_2$: [14]

```
                firstPlateau([X,Y],[X,Y],[]) ⇐ X=Y
        NFI: LA(=)      ↓      {Y/X}
              firstPlateau([X,X],[X,X],[]) ⇐
          DCI: C3       ↓      {}
        [X,X]=[?_,?_|?_] ∧ [X,X]=[?HL|?TL] ∧
              firstPlateau(?TL,?TP,?TS) ∧
      [X,X]=[?HL|?TP] ∧ []=?TS ∧ ?TP=[?HL|?_] ⇐
      5 × DCI: LA(=)    ↓      {HL/X, TL/[X], TP[X], TS/[]}
              firstPlateau([X],[X],[]) ⇐
          DCI: P1       ↓      {}
                        ❑
```

This derivation succeeds. ♦

For a non-recursive property P_i, a normalized successful derivation is an instance of the following derivation schema: [15]

```
                        r(t) ⇐ B
                NFI      ↓      λ
                      r(t)λ ⇐ Q
DCI: clause Ck from LA(r) ↓      {}
                    S ∧ r(s) ⇐ Q
                DCI      ↓      σ
                     r(s)σ ⇐ Qσ
        DCI: P(r) \ {Pi}  ↓      μ
                      T  ⇐ Qσμ
            DCI, Sim     ↓      ρ
                       ⇐ Qσμρ
```

where:

- B, Q, S, T are (possibly empty) conjunctions of atoms with primitive predicates,
- $\boldsymbol{s}, \boldsymbol{t}$ are vectors of terms,
- $\lambda, \sigma, \mu, \rho$ are mgus.

The relevant part of the computed answer substitution, denoted φ, is the composition $\sigma\mu\rho$.

14. In sample derivations, the selected atom(s) for the next inference(s) is (are) written in boldface (because there is no ambiguity with the vector convention).
15. In derivation schemas, the selected atom(s) for the next inference(s) is (are) underlined.

If clause C_k is not recursive, then everything related to ***r(s)*** may simply be ignored. A slight variation of this schema can be established for recursive properties.

We are now able to state a theorem that shows a practical way of verifying the completeness of a logic algorithm wrt a property set.

Theorem 9-1: *A logic algorithm LA(r) is complete (according to the new criterion* (1)*) wrt a property set $\mathcal{P}(r)$ iff, for every property P_i in $\mathcal{P}(r)$, there exists a successful derivation (via the NFI, DCI, and Sim rules) of $\mathcal{D}_i \vdash P_i$, where theory $\mathcal{D}_i$ is defined as above.*

Proof 9-1: The theorem directly follows from the soundness and completeness of extended execution wrt the Clark completion semantics [Kanamori 86]. ❑

The problem statement of the Proofs-as-Programs Method requires that *LA(r)* be complete wrt $\mathcal{P}(r)$, and this, in retrospective, according to the new criterion (1). Indeed, there wouldn't exist any complete syntactic specialization otherwise. But in practice, it is often impossible to know beforehand whether this constraint is satisfied or not. Theorem 9-1 thus shows a way of verifying this simultaneously with the attempts at enhancement, as captured in the following definition:

Definition 9-5: The Proofs-as-Programs Method *succeeds* iff *LA(r)* is proven (by extended execution) to be complete wrt $\mathcal{P}(r)$. It *fails* otherwise.

So far for the proof aspects of the Proofs-as-Programs Method. Let's now turn to the condition extraction aspects.

9.2.2 The Extraction of Conditions

The fundamental observation of condition extraction is that the final goal of a successful derivation may have a non-empty hypothesis. So the idea is to use that hypothesis (even if it is empty, that is *true*) together with the computed answer substitution in order to build a condition that, if added to the original logic algorithm, would then give rise to an unconditionally successful derivation.

More precisely, let C_k be the k^{th} clause generated from *LA(r)*. Assume C_k is as follows:

$$\mathtt{r(U)} \leftarrow \exists \mathtt{W}\ \mathtt{B_k[U]}$$

where:

- ***U*** is the vector of universal variables in the k^{th} disjunct of *LA(r)*,
- ***W*** is the vector of existential variables in the k^{th} disjunct of *LA(r)*.

In case of a successful derivation whose first DCI resolution was based on C_k, a (new) predicate q_k is partially defined by the definite program clause:

$$q_k(\boldsymbol{t},\boldsymbol{W})\varphi \leftarrow Q\varphi \qquad (3)$$

where:

- ***t*** is the vector of terms in the head of P_i,
- φ is the computed answer substitution,
- $Q\varphi$ is the hypothesis of the last goal of the derivation.

Note that such a definite program clause is defined in terms of primitive predicates only. Indeed, $Q\varphi$ originally is the body of property P_i (remember that properties are defined in terms of primitives only), and atoms can be deleted from it (by use of the *NFI* and *Sim* rules), and atoms with primitive predicates can be added to it (by use of the *NFI* rule). An important feature is thus that these definite program clauses can't be recursive.

Before proceeding with the theoretical considerations, let's illustrate all this on a sample condition extraction.

Example 9-3: The derivation of Example 9-2 succeeds, and a new predicate, say $discFirstPlateau_3$, is partially defined as follows:

```
discFirstPlateau3(L,P,S,HL,TL,TP,TS) ←
                          L=[X,X],P=[X,X],S=[],
                          HL=X,TL=[X],TP=[X],TS=[]        ♦
```

Once that all successful derivations of all formulas (2) have been computed, and provided the method succeeds, *LA'(r)* is obtained from *LA(r)* by adding an atom $q_k(\boldsymbol{U},\boldsymbol{W})$ to its k^{th} disjunct, for every newly created predicate q_k.

Similarly, a set $\mathcal{Z}$ of logic algorithms $LA(q_k)$ is obtained by translating the definite procedures of the q_k into the logic algorithm language. The used translation rules are the reverse of those used above for achieving the applicability of extended execution: they amount to computing the Clark completion of the definite procedures of the q_k. These logic algorithms are by construction non-recursive, so they may be unfolded into *LA'(r)*.

The Proofs-as-Programs Method is deterministic because the results of all successful derivations are collected before computing $\mathcal{Z}$ therefrom. The method is thus independent of any ordering of disjuncts within *LA(r)*, or of properties within $\mathcal{P}(r)$. The method may fail, as conveyed in Definition 9-5. However, nothing can be said about the synthesized logic algorithms in terms of determinism or finiteness, because nothing is known about the properties.

Establishing the complexity of the method is quite intricate, unless some assumptions are made. Thus, assume that properties are non-recursive, and that all primitives used in $\mathcal{P}(r)$ are deterministic, and let c be the number of disjuncts in *LA(r)*, p be the number of properties of $\mathcal{P}(r)$, and t be the number of recursive atoms in *LA(r)*. There are p proofs to be made. Each proof-tree has only two choice-points, namely the *DCI* resolution of the head of the initial goal, where there are c possibilities, and the *DCI* resolution of the t recursive atoms, where there are $p-1$ possibilities, all other proof steps being deterministic. Each proof tree has thus size $O(cp^t)$. Hence, the time complexity of the method is $O(cp^{t+1})$. This assumes that there is a fixed maximum number of atoms for the definitions of the used primitives. Indeed, if that number is a function of c, t, or p, then this complexity analysis doesn't hold.

9.3 Correctness

The following correctness theorem can now be established.

Theorem 9-2: *LA'(r)* is a syntactic specialization of *LA(r)* that is totally correct wrt $\mathcal{P}(r)$.

Proof 9-2: Let $\mathcal{T}_i'$ be defined like $\mathcal{T}_i$, but using *LA'(r)* rather than *LA(r)*. Suppose we now prove that each property P_i is a logical consequence of its theory $\mathcal{T}_i'$. Without loss of generality, we can assume that the previous derivations of $\mathcal{D}_i \vdash P_i$ are prefixes of the new derivations of $\mathcal{D}_i' \vdash P_i$ (namely by a relaxation of the recommendations above for the computation rules and for the normalization of proofs). Each new derivation then eventually yields a goal whose conclusion only involves a $\mathtt{q_k}$ atom, as shown by the revised derivation schema:

$$
\begin{array}{rl}
 & \mathtt{r(t)} \Leftarrow \underline{\mathtt{B}} \\
\textit{NFI} & \downarrow \quad \lambda \\
 & \underline{\mathtt{r(t)}\lambda} \Leftarrow \mathtt{Q} \\
\textit{DCI}\text{: clause } C'_k \text{ from } \textit{LA'(r)} & \downarrow \quad \{\} \\
 & \underline{\mathtt{S}} \wedge \mathtt{r(s)} \wedge \mathtt{q_k(t,W)} \Leftarrow \mathtt{Q} \\
 & \textit{DCI} \quad \downarrow \quad \sigma \\
 & \underline{\mathtt{r(s)}\sigma} \wedge \mathtt{q_k(t,W)}\sigma \Leftarrow \mathtt{Q}\sigma \\
\textit{DCI}\text{: } \mathcal{P}(r) \setminus \{P_i\} & \downarrow \quad \mu \\
 & \underline{\mathtt{T}} \wedge \mathtt{q_k(t,W)}\sigma\mu \Leftarrow \underline{\mathtt{Q}\sigma\mu}
\end{array}
$$

$$\begin{array}{rcl} \textit{DCI, Sim} & \downarrow & \rho \\ q_k\texttt{(t,W)}\sigma\mu\rho & \Leftarrow & Q\sigma\mu\rho \end{array}$$

Replacing $\sigma\mu\rho$ by φ, this last goal may be rewritten more compactly, and the revised derivation schema continues as follows:

$$\begin{array}{rcl} \underline{q_k\texttt{(t,W)}\varphi} & \Leftarrow & Q\varphi \\ \textit{DCI}\text{: clause (3)} & \downarrow & \{\} \\ \underline{Q\varphi} & \Leftarrow & \underline{Q\varphi} \\ \textit{Sim} & \downarrow & \{\} \\ \multicolumn{3}{c}{\blacksquare} \end{array}$$

Every new derivation thus succeeds as well. By soundness and completeness of extended execution wrt the Clark completion semantics [Kanamori 86], the property set $\mathcal{P}(r)$ is thus also logical consequence of *LA'(r)*. Hence *LA'(r)* is complete wrt $\mathcal{P}(r)$.

Note that *LA'(r)* is certainly the most specific syntactic specialization of *LA(r)* that the Proofs-as-Programs Method can produce: starting from *LA'(r)*, a renewed application of that method would, as the proof above indicates, lead to a logic algorithm that is syntactically equivalent to *LA'(r)*. The Proofs-as-Programs Method is thus idempotent, and *LA'(r)* is partially correct wrt $\mathcal{P}(r)$, that is syntactically equivalent to $\mathcal{P}(r)$ expressed as a logic algorithm.

By construction, *LA'(r)* is a syntactic specialization of *LA(r)*. ❑

9.4 Illustration

Let's illustrate the described method on two sample problems.

Example 9-4: As a reminder, the *compress(L,C)* procedure succeeds iff *C* is a compact list of $<v_i,c_i>$ couples, such that the i^{th} plateau of list *L* has c_i elements equal to v_i. The logic algorithm *LA(compress)* that is to be specialized is:

```
compress(L,C) ⇔
    L=[]        ∧ C=[]
  ∨ L=[_|_]     ∧ L=[HL|TL]
                ∧ compress(TL,TC)
                ∧ C=[HL,1|TC]
  ∨ L=[_|_]     ∧ L=[HL|TL]
                ∧ compress(TL,TC)
                ∧ C=[HL,s(s(N))|TTC] ∧ TC=[HL,s(N)|TTC]
```

The corresponding definite program is:

```
compress(L,C) ← L=[],C=[]                                    (C1)
compress(L,C) ← L=[_|_],
                L=[HL|TL],
                compress(TL,TC),
                C=[HL,1|TC]                                  (C2)
compress(L,C) ← L=[_|_],
                L=[HL|TL],
                compress(TL,TC),
                C=[HL,s(s(N))|TTC],TC=[HL,s(N)|TTC]          (C3)
```

Finally, let the properties be:

```
compress([],[])                                              (E1)
compress([X],[X,1])                                          (P1)
```

```
compress([X,Y],[X,2]) ⇐ X=Y                                    (P2)
compress([X,Y],[X,1,Y,1]) ⇐ X≠Y                                (P3)
```

The objective is to attempt to prove, by extended execution, that each property P_i is a logical consequence of theory $\mathcal{T}_i$.

We skip the proof of property E_1, as it is trivial, and as it leads to the extraction of a condition that is redundant wrt the existing atoms.

Let's pursue with P_1, and use clause C_1 for the first DCI resolution:

compress([X],[X,1]) ⇐
DCI: C_1 ↓ {}
[X]=[] ∧ [X,1]=[] ⇐

This derivation fails, and no extraction is performed.

We continue with clause C_2 for the first DCI resolution:

compress([X],[X,1]) ⇐
DCI: C_2 ↓ {}
[X]=[?_|?_] ∧ **[X]=[?HL|?TL]** ∧
compress(?TL,?TC) ∧ **[X,1]=[?HL,1|?TC]** ⇐
3 × *DCI*: *LA*(=) ↓ {*HL*/*X*, *TL*/[], *TC*/[]}
compress([],[]) ⇐
DCI: E_1 ↓ {}
❑

This derivation succeeds. A clause for a new predicate $discCompress_2$ is defined as follows:

```
discCompress2(L,C,HL,TL,TC) ←
                        L=[X],C=[X,1],
                        HL=X,TL=[],TC=[]
```

We continue with clause C_3 for the first DCI resolution:

compress([X],[X,1]) ⇐
DCI: C_3 ↓ {}
[X]=[?_|?_] ∧ [X]=[?HL|?TL] ∧ compress(?TL,?TC) ∧
[X,1]=[?HL,s(s(?N))|?TTC] ∧ ?TC=[?HL,s(?N)|?TTC] ⇐

This derivation fails, and no extraction is performed.

We pursue with P_2, and restart from clause C_1 for the first DCI resolution:

compress([X,Y],[X,2]) ⇐ **X=Y**
NFI: *LA*(=) ↓ {*Y*/*X*}
compress([X,X],[X,2]) ⇐
DCI: C_1 ↓ {}
[X,X]=[] ∧ [X,2]=[] ⇐

This derivation fails, and no extraction is performed.

We continue with clause C_2 for the first DCI resolution:

compress([X,Y],[X,2]) ⇐ **X=Y**
NFI: *LA*(=) ↓ {*Y*/*X*}
compress([X,X],[X,2]) ⇐
DCI: C_2 ↓ {}

```
[X,X]=[?_|?_] ∧ [X,X]=[?HL|?TL] ∧
compress(?TL,?TC) ∧ [X,2]=[?HL,1|?TC] ⇐
```

This derivation also fails, and no extraction is performed.

We continue with clause C_3 for the first DCI resolution:

```
compress([X,Y],[X,2]) ⇐ X=Y
```
NFI: *LA*(=) ↓ {*Y*/*X*}
```
compress([X,X],[X,2]) ⇐
```
DCI: C_3 ↓ {}
```
[X,X]=[?_|?_] ∧ [X,X]=[?HL|?TL] ∧ compress(?TL,?TC) ∧
[X,2]=[?HL,s(s(?N))|?TTC] ∧ ?TC=[?HL,s(?N)|?TTC] ⇐
```
2 × *DCI*: *LA*(=) ↓ {*HL*/*X*, *TL*/[*X*], *N*/0, *TTC*/[], *TC*/[*X*,1]}
```
compress([X],[X,1]) ⇐
```
DCI: P_1 ↓ {}

❑

This derivation succeeds. A new predicate $discCompress_3$ is partially defined as follows:

```
discCompress3(L,C,HL,TL,TC,N,TTC) ←
                    L=[X,X],C=[X,2],
                    HL=X,TL=[X],TC=[X,1],
                    N=0,TTC=[]
```

We pursue with P_3, and restart from clause C_1 for the first DCI resolution:

```
compress([X,Y],[X,1,Y,1]) ⇐ X≠Y
```
DCI: C_1 ↓ {}
```
[X,Y]=[] ∧ [X,1,Y,1]=[] ⇐ X≠Y
```

This derivation fails, and no extraction is performed.

We continue with clause C_2 for the first DCI resolution:

```
compress([X,Y],[X,1,Y,1]) ⇐ X≠Y
```
DCI: C_2 ↓ {}
```
[X,Y]=[?_|?_] ∧ [X,Y]=[?HL|?TL] ∧
compress(?TL,?TC) ∧ [X,1,Y,1]=[?HL,1|?TC] ⇐ X≠Y
```
3 × *DCI*: *LA*(=) ↓ {*HL*/*X*, *TL*/[*Y*], *TC*/[*Y*,1]}
```
compress([Y],[Y,1]) ⇐ X≠Y
```
DCI: P_1 ↓ {}
```
⇐ X≠Y
```

This derivation succeeds. Another clause for predicate $discCompress_2$ is created as follows:

```
discCompress2(L,C,HL,TL,TC) ←
                    L=[X,Y],C=[X,1,Y,1],
                    HL=X,TL=[Y],TC=[Y,1],
                    X≠Y
```

We continue with clause C_3 for the first DCI resolution:

```
compress([X,Y],[X,1,Y,1]) ⇐ X≠Y
```
DCI: C_3 ↓ {}

```
    [X,Y]=[?_|?_] ∧ [X,Y]=[?HL|?TL] ∧ compress(?TL,?TC) ∧
[X,1,Y,1]=[?HL,s(s(?N))|?TTC] ∧ ?TC=[?HL,s(?N)|?TTC]  ⇐  X≠Y
```

This derivation fails, and no extraction is performed.

There is no other property. There are no alternative derivations. So *LA'(compress)* is defined as follows, after adding two atoms to *LA(compress)*:

```
compress(L,C) ⇔
    L=[]       ∧ C=[]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ discCompress2(L,C,HL,TL,TC)
               ∧ compress(TL,TC)
               ∧ C=[HL,1|TC]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ discCompress3(L,C,HL,TL,TC,N,TTC)
               ∧ compress(TL,TC)
               ∧ C=[HL,s(s(N))|TTC] ∧ TC=[HL,s(N)|TTC]
```

where the procedures of the new predicates are rewritten as a set $\mathcal{Z}$ of logic algorithms:

```
discCompress2(L,C,HL,TL,TC) ⇔
                  L=[X] ∧ C=[X,1]
                ∧ HL=X ∧ TL=[] ∧ TC=[]
              ∨   L=[X,Y] ∧ C=[X,1,Y,1]
                ∧ HL=X ∧ TL=[Y] ∧ TC=[Y,1]
                ∧ X≠Y
discCompress3(L,C,HL,TL,TC,N,TTC) ⇔
                  L=[X,X] ∧ C=[X,2]
                ∧ HL=X ∧ TL=[X] ∧ TC=[X,1]
                ∧ N=0 ∧ TTC=[]
```

Suppose now that some application-specific heuristics (see Section 13.3.2) suggest that:

- both conditions may be generalized by projection onto their 2nd to 4th parameters;
- the values of the *C* parameter are irrelevant in both conditions;
- the *TL* parameter should vary over its entire domain in both conditions.

This transforms $\mathcal{Z}$ into $\mathcal{Z}$', which is as follows (after some additional rewriting):

```
discCompress2(L,C,HL,TL,TC) ⇔
                      TL=[]
                    ∨ TL=[HTL|_] ∧ HL≠HTL
discCompress3(L,C,HL,TL,TC,N,TTC) ⇔
                      TL=[HTL|_] ∧ HL=HTL
```

If we unfold *LA'(compress)* using $\mathcal{Z}$', then we obtain *LA"(compress)*:

```
compress(L,C) ⇔
    L=[]       ∧ C=[]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ (TL=[]) ∨ (TL=[HTL|_] ∧ HL≠HTL)
               ∧ compress(TL,TC)
               ∧ C=[HL,1|TC]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ TL=[HTL|_] ∧ HL=HTL
               ∧ compress(TL,TC)
               ∧ C=[HL,s(s(N))|TTC] ∧ TC=[HL,s(N)|TTC]
```

♦

Example 9-5: As a reminder, the *firstPlateau(L,P,S)* relation holds iff P is the first maximal plateau of the non-empty list L, and list S is the corresponding suffix of L. The logic algorithm *LA(firstPlateau)* that is to be specialized is:

```
firstPlateau(L,P,S) ⇔
    L=[_]      ∧ P=L ∧ S=[] ∧ L=[_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ firstPlateau(TL,TP,TS)
              ∧ P=[HL|TP] ∧ S=TS ∧ TP=[HL|_]
```

The corresponding definite program is (after performing an obvious simplification in C_1):

```
firstPlateau(L,P,S) ←  L=[_],P=L,S=[]                    (C1)
firstPlateau(L,P,S) ←  L=[_,_|_],
                       L=[HL|TL],
                       P=[HL],S=TL,TL=[_|_]              (C2)
firstPlateau(L,P,S) ←  L=[_,_|_],
                       L=[HL|TL],
                       firstPlateau(TL,TP,TS),
                       P=[HL|TP],S=TS,TP=[HL|_]          (C3)
```

Finally, let the properties be:

```
firstPlateau([X] , [X] , [] )                            (P1)
firstPlateau([X,Y],[X,Y],[] ) ⇐ X=Y                      (P2)
firstPlateau([X,Y],[X] , [Y]) ⇐ X≠Y                      (P3)
```

The objective is to attempt to prove, by extended execution, that each property P_i is a logical consequence of theory $\mathcal{T}_i$.

Let's start with P_1, and use clause C_1 for the first DCI resolution:

firstPlateau([X],[X],[]) ⇐
DCI: C_1 ↓ {}
[X]=[?_] ∧ **[X]=[X]** ∧ **[]=[]** ⇐
$3 \times$ *DCI*: *LA*(=) ↓ {}
❑

This derivation succeeds. A new predicate $discFirstPlateau_1$ is partially defined as follows:

```
discFirstPlateau1(L,P,S) ←
                          L=[X],P=[X],S=[]
```

We continue with clause C_2 for the first DCI resolution:

firstPlateau([X],[X],[]) ⇐
DCI: C_2 ↓ {}
[X]=[?_,?_|?_] ∧ [X]=[?HL|?TL] ∧
[X]=[?HL] ∧ []=?TL ∧ ?TL=[?_|?_] ⇐

This derivation fails, and no extraction is performed.

We continue with clause C_3 for the first DCI resolution:

firstPlateau([X],[X],[]) ⇐
DCI: C_3 ↓ {}

[X]=[?_,?_|?_] ∧ [X]=[?HL|?TL] ∧ firstPlateau(?TL,?TP,?TS) ∧
[X]=[?HL|?TP] ∧ []=?TS ∧ ?TP=[?HL|?_] ⇐

This derivation also fails, and no extraction is performed.

We pursue with P_2, and restart from clause C_1 for the first DCI resolution:

firstPlateau([X,Y],[X,Y],[]) ⇐ **X=Y**

NFI: *LA*(=) ↓ {*Y*/*X*}

firstPlateau([X,X],[X,X],[]) ⇐

DCI: C_1 ↓ {}

[X,X]=[?_] ∧ [X,X]=[X,X] ∧ []=[] ⇐

This derivation fails, and no extraction is performed.

We continue with clause C_2 for the first DCI resolution:

firstPlateau([X,Y],[X,Y],[]) ⇐ **X=Y**

NFI: *LA*(=) ↓ {*Y*/*X*}

firstPlateau([X,X],[X,X],[]) ⇐

DCI: C_2 ↓ {}

[X,X]=[?_,?_|?_] ∧ [X,X]=[?HL|?TL] ∧
[X,X]=[?HL] ∧ []=?TL ∧ ?TL=[?_|?_] ⇐

This derivation also fails, and no extraction is performed.

We continue with clause C_3 for the first DCI resolution. This is shown in Example 9-2, where the derivation succeeds, and a new predicate *discFirstPlateau*$_3$ is defined as follows:

discFirstPlateau$_3$(L,P,S,HL,TL,TP,TS) ←
L=[X,X],P=[X,X],S=[],
HL=X,TL=[X],TP=[X],TS=[]

We pursue with P_3, and restart from clause C_1 for the first DCI resolution:

firstPlateau([X,Y],[X],[Y]) ⇐ X≠Y

DCI: C_1 ↓ {}

[X,Y]=[?_] ∧ [X]=[X,Y] ∧ [Y]=[] ⇐ X≠Y

This derivation fails, and no extraction is performed.

We continue with clause C_2 for the first DCI resolution:

firstPlateau([X,Y],[X],[Y]) ⇐ X≠Y

DCI: C_2 ↓ {}

[X,Y]=[?_,?_|?_] ∧ **[X,Y]=[?HL|?TL]** ∧
[X]=[?HL] ∧ **[Y]=?TL** ∧ **?TL=[?_|?_]** ⇐ X≠Y

5 × *DCI*: *LA*(=) ↓ {*HL*/*X*, *TL*/[*Y*]}

⇐ X≠Y

This derivation succeeds. A new predicate *discFirstPlateau*$_2$ is partially defined as follows:

discFirstPlateau$_2$(L,P,S,HL,TL) ←
L=[X,Y],P=[X],S=[Y],
HL=X,TL=[Y],
X≠Y

We continue with clause C_3 for the first DCI resolution:

firstPlateau([X,Y],[X],[Y]) ⇐ X≠Y

DCI: C_3 ↓ {}

[X,Y]=[?_,?_|?_] ∧ **[X,Y]=[?HL|?TL]** ∧
firstPlateau(?TL,?TP,?TS) ∧
[X]=[?HL|?TP] ∧ **[Y]=?TS** ∧ ?TP=[?HL|?_] ⇐ X≠Y

$4 \times$ *DCI*: *LA*(=) ↓ {*HL*/*X*, *TL*/[*Y*], *TP*[], *TS*/[*Y*]}

firstPlateau([Y],[],[Y]) ∧ **[]=[X|?_]** ⇐ X≠Y

This derivation fails, and no extraction is performed.

There is no other property. There are no alternative derivations. So *LA'(firstPlateau)* is defined as follows, after adding three atoms to *LA(firstPlateau)*:

```
firstPlateau(L,P,S) ⇔
    L=[_]      ∧ P=L ∧ S=[] ∧ L=[_]
               ∧ discFirstPlateau1(L,P,S)
  ∨ L=[_,_|_] ∧ L=[HL|TL]
               ∧ discFirstPlateau2(L,P,S,HL,TL)
               ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
               ∧ discFirstPlateau3(L,P,S,HL,TL,TP,TS)
               ∧ firstPlateau(TL,TP,TS)
               ∧ P=[HL|TP] ∧ S=TS ∧ TP=[HL|_]
```

where the procedures of the new predicates are rewritten as a set $\mathcal{Z}$ of logic algorithms:

```
discFirstPlateau1(L,P,S) ⇔
                L=[X] ∧ P=[X] ∧ S=[]

discFirstPlateau2(L,P,S,HL,TL) ⇔
                L=[X,Y] ∧ P=[X] ∧ S=[Y]
              ∧ HL=X ∧ TL=[Y]
              ∧ X≠Y

discFirstPlateau3(L,P,S,HL,TL,TP,TS) ⇔
                L=[X,X] ∧ P=[X,X] ∧ S=[]
              ∧ HL=X ∧ TL=[X] ∧ TP=[X] ∧ TS=[]
```

Suppose now that some application-specific heuristics (see Section 13.3.2) suggest that:

- the last two conditions may be projected onto their 2nd to 5th parameters;
- the values of the *P* and *S* parameters are irrelevant in all three conditions;
- the *TL* parameter should vary over its entire domain in the last two conditions.

This transforms $\mathcal{Z}$ into $\mathcal{Z}$', which is as follows (after some additional rewriting):

```
discFirstPlateau1(L,P,S) ⇔
                L=[_]

discFirstPlateau2(L,P,S,HL,TL) ⇔
                TL=[HTL|_] ∧ HL≠HTL

discFirstPlateau3(L,P,S,HL,TL,TP,TS) ⇔
                TL=[HTL|_] ∧ HL=HTL
```

If we unfold *LA'(firstPlateau)* using $\mathcal{Z}$', then we obtain *LA"(firstPlateau)*:

```
firstPlateau(L,P,S) ⇔
    L=[_]      ∧ P=L ∧ S=[] ∧ L=[_] ∧ L=[_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ TL=[HTL|_] ∧ HL≠HTL
              ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ TL=[HTL|_] ∧ HL=HTL
              ∧ firstPlateau(TL,TP,TS)
              ∧ P=[HL|TP] ∧ S=TS ∧ TP=[HL|_]          ♦
```

9.5 Future Work

This method can be easily extended to handle negated primitive predicates since they are resolved according to their semantics. Moreover, the constraint that *LA(r)* should be expressed solely in terms of primitive predicates and possibly of the *r/n* predicate can be easily overcome, namely by considering sets of logic algorithms rather than a single logic algorithm.

A more sophisticated extension would be the handling of proofs-by-induction. This requires additional rules of inference, such as those of [Kanamori and Fujita 86] or [Fribourg 91a]. Such proofs are sometimes necessary, as shown in Example 14-6.

9.6 Related Work

The research related to our Proofs-as-Programs Method can be separated into two parts: the motivation for the terminology used in naming this method, as well as the discussion of other methods that solve the same (or a similar) problem as the one defined here.

Our Proofs-as-Programs Method is not like the other methods of program extraction from proofs (see Chapter 2), since the program is here extracted from the unique final results of several proofs, rather than from multiple intermediate steps of a single proof. What justifies our naming the method as such is a broader interpretation of the proofs-as-programs paradigm: it covers any method of extracting a computational content from a proof.

In terms of methods that solve the same (or a similar) problem as the one tackled by our Proofs-as-Programs Method, there first is the method described by [Smith 82]: given two formulas F and G, the objective is to find the weakest precondition P such that G follows from $F \wedge P$. Taking F as the initial logic algorithm, G as its property set, and $F \wedge P$ as the final logic algorithm, this problem is a sub-case of ours, because F, G, and P are here sets of formulas. Smith's proofs are performed by natural deduction.

In explanation-based generalization, there is some research on failure recovery (such as [Gupta 87]), which is quite similar in spirit to the derivation of weakest pre-conditions.

Another related research is about *deductive debugging*, as reported by [Dershowitz and Lee 87]. In the context of logic program debugging, suppose that, upon execution of a test-case, a bug has been discovered, located, and classified as a "missing condition" bug. Their *automated program debugger* (*apd*) then adds the missing condition by extracting it from a failed proof that the program satisfies its (complete) specification. Their method seems quite close to ours, except for the used proof-theoretic framework.

The problem of *belief updating from integrity constraints and queries* is introduced by [De Raedt and Bruynooghe 92] as a generalization of the problems of intensional knowledge-base updating and incremental concept-learning. Given a knowledge-base *KB*, an integrity constraint theory *IT* that is satisfied by *KB*, an integrity constraint *IC*, and an oracle *O* that is willing to answer existential and membership questions, the problem of belief updating amounts to finding another knowledge base *KB'* that satisfies *IT*+*IC*, and that is obtained

by asserting/retracting any clause/fact to/from *KB*, possibly via asking questions to *O*. If we abstract away the differences in the chosen languages, this problem statement is very close to the formulation of our Proofs-as-Programs problem: the logic algorithm is replaced by a knowledge-base, properties become integrity constraints, and there is an oracle now. But from these very problem statements, one can also already detect a first major difference: their integrity constraints are incrementally presented and verified, whereas our properties are presented and verified all-at-once. Other main differences lie in the solutions to the two problems. The belief updating problem is solved by a simple adaptation of Shapiro's *Model Inference System* (MIS, see Section 3.4.1) [Shapiro 82]: integrity constraints are a generalization of examples. Also, integrity constraints are only used for verifying the results of the learning, and not for actually constructing these results. This means that the clause specialization algorithms of *MIS* are still only based on the examples, that is a subset of the provided integrity constraints. The Proofs-as-Programs Method is more powerful in that it actually extracts complex information from properties so as to constructively use it in the design of the target logic algorithm.

It is also interesting to directly compare our Proofs-as-Programs Method with MIS. Both "debug" a given statement (logic algorithm, respectively logic program) such that it satisfies a given set of constraints (properties, respectively positive and negative examples), using some theorem proving technique (extended execution, respectively SLD resolution) for the diagnosis. In case of incompleteness (failure to prove a positive constraint), the former adds some atom(s) to some disjunct of the logic algorithm, whereas the latter adds a clause to the logic program. In case of incorrectness (proof of a negative constraint), the latter deletes some clause from the logic program, whereas this case is impossible with the former (there are no negative properties). In case of potential divergence (failure to prove a constraint within preset resource limits), the latter deletes some clause from the logic program, whereas this case is impossible with the former (by construction, as explained above).

9.7 Conclusion

In this chapter, we have presented the Proofs-as-Programs Method, which adds atoms to (and hence specializes) a logic algorithm so that it is complete wrt a set of given properties. The added atoms are extracted from the proof that the initial logic algorithm is already complete wrt these properties.

10 The Most-Specific-Generalization Method

In this chapter, we develop the Most-Specific-Generalization (MSG) Method, which, within a restricted setting, inductively infers a logic algorithm from examples. This method is part of our tool-box of methods for instantiating the predicate variables of a schema. First, in Section 10.1, we define the concept of most-specific-generalization. Then, in Section 10.2, we state the objective of the MSG Method, and introduce some other preliminary terminology. We proceed by increasing difficulty, and start with the simplest form the problem can take. Thus, in Section 10.3, we first discuss the case where every example is a ground atom. Then, in Section 10.4, we study the case where non-ground examples, called *general examples*, are allowed. General examples have disjunctions and existential variables, and are thus different from properties. Future work and related work are discussed in Section 10.5 and Section 10.6, respectively, before drawing some conclusions in Section 10.7.

10.1 Most-Specific-Generalization of Terms

The concept of most specific generalization (msg) was introduced independently by [Plotkin 70] and [Reynolds 70]. We focus our attention to terms and atoms, rather than to wffs.

Definition 10-1: Term s is *less general* than term t (denoted $s \leq t$) iff there is a substitution σ, such that $s = t\sigma$.

The relation $\leq/2$ forms a complete lattice on the term set $\mathcal{T}$(modulo variable renaming) to which a least element has been added [Lassez *et al*. 87]. The glb operator computes the gci (greatest common instance) of two terms (by a unification algorithm, yielding the mgu). The lub operator computes the msg of two terms (by an anti-unification algorithm). The msg of two terms thus always exists, and is unique up to variable renaming.

Here follows another, constructive, definition of the concept of msg:

Definition 10-2: Let s and t designate two terms $f(s_1,s_2,\ldots,s_m)$ and $g(t_1,t_2,\ldots,t_n)$, respectively, where $m \geq 0, n \geq 0$. The *most specific generalization* of s and t, denoted $msg(s,t)$, is defined as follows:

- if $f/m = g/n$, then $msg(s,t) = f(msg(s_1,t_1),msg(s_2,t_2),\ldots,msg(s_m,t_m))$;
- otherwise, $msg(s,t) = inj(s,t)$;

where inj is an injection from $\mathcal{T} \times \mathcal{T}$ into $\mathcal{V}$(the set of variable symbols).

Example 10-1: The msg of the terms $f(a,b,X,Y,X,c)$ and $f(a,c,d,Z,d,b)$ is $f(a,K,L,M,L,N)$.

Definition 10-3: The *most specific generalization* of a non-empty set $\mathcal{S}$ of terms, denoted $msg(\mathcal{S})$, is defined as follows:

- $msg(\{t\}) = t$;
- $msg(\mathcal{T} \cup \{t\}) = msg(msg(\mathcal{T}),t)$, where $\mathcal{T}$ is a non-empty set of terms;

where the difference between $msg/1$ and $msg/2$ should be noted.

Definition 10-4: Let s and t designate two atoms $r(s_1,s_2,\ldots,s_n)$ and $r(t_1,t_2,\ldots,t_n)$. The *most specific generalization* of s and t, denoted $msg(s,t)$, is the atom $r(m_1,m_2,\ldots,m_n)$, where $<m_1,m_2,\ldots,m_n> = msg(<s_1,s_2,\ldots,s_n>,<t_1,t_2,\ldots,t_n>)$.

The msg of a non-empty set of atoms is defined in the same way as the msg of a non-empty set of terms. The overloading of the *msg* operator (different arities, different types of parameters) should pose no problem to the reader, especially that the different concepts are closely related anyway, and that we only use the msg of atoms with the same predicate symbol.

10.2 Objective and Terminology

Let's close in now on the MSG Method. Intuitively, its objective is to infer a logic algorithm of a predicate *r*/*n*, given a finite set of examples of *r*/*n*. [16] The method should be applicable if the intended relation $\mathcal{R}$ (from which the examples are extracted) can be expressed by a logic algorithm that is defined solely in terms of the =/2 primitive (hence is non-recursive, among others). This is feasible iff, in the intended relation, some parameters are somehow syntactically constructed from some other parameters.

In a first approximation, we suppose that $\mathcal{R}$ is ternary, and that its third parameter is syntactically constructed from its second parameter, while the first parameter may or may not be used in this construction.

Let's first illustrate the objective on a sample problem.

Example 10-2: Given the example set:

```
E(pcCompress)={ pcCompress(a,[],[a,1])                    (E2)
                pcCompress(b,[b,1],[b,2])                 (E3)
                pcCompress(c,[d,1],[c,1,d,1])             (E4)
                pcCompress(e,[e,2],[e,3])                 (E5)
                pcCompress(f,[f,1,g,1],[f,2,g,1])         (E6)
                pcCompress(h,[i,2],[h,1,i,2])             (E7)
                pcCompress(j,[k,1,m,1],[j,1,k,1,m,1]) }   (E8)
```

the MSG Method could infer the following version of *LA(pcCompress)*:

```
pcCompress(HL,TC,C) ⇔
    C=[HL,1|TC]
  v C=[HL,s(s(N))|TTC] ∧ TC=[HL,s(N)|TTC]
```

which is indeed complete wrt *E(pcCompress)* and all "similar" examples. ♦

Let's now formally define what it means for an atom to satisfy the parameter construction constraint mentioned above.

Definition 10-5: An atom $r(t_1,t_2,t_3)$ is *admissible* iff the following two conditions hold:

$$cons(t_2) \subseteq cons(t_3)$$
$$vars(t_2) \subseteq vars(t_3).$$

In other word, the constituents of t_2 must be included in the constituents of t_3, where the *constituents* of a term are the constants and variables occurring in that term.

Definition 10-6: A set $\mathcal{A}(r)$ of atoms is *admissible* iff all atoms of $\mathcal{A}(r)$ are admissible.

Example 10-3: The set *E(pcCompress)* above is admissible.

We also need a criterion for verifying whether several atoms construct their third parameters from their second parameters in the same way.

Definition 10-7: Let *s, t* be two atoms of predicate *r*. Then *s* is *compatible* with *t* iff *msg(s,t)* is admissible. We also say that *s* and *t* are compatible.

Definition 10-8: Let $\mathcal{A}(r)$ be a set of atoms. Then $\mathcal{A}(r)$ is *compatible* iff the atoms of $\mathcal{A}(r)$ are pairwisely compatible.

Example 10-4: Take the data of Example 10-2 again. Then E_6 is compatible with E_1, because the msg *pcCompress(A,T,*[*A*,1|*T*]) is admissible. But E_7 is not compatible with E_2, because the msg *pcCompress(A,*[*B,s*(0)|*T*],[*A,s(M)*|*U*]) is not admissible.

16. In the sequel, we shall actually distinguish between (ground) examples, as in Definition 6-1, and general examples (to be defined).

The compatibility relation is reflexive and symmetric, but not transitive. Compatible sets are thus sets over which the compatibility relation is total.

Example 10-5: The atom $r(a,[b],[a,b])$ is compatible with $r(c,[d,d],[c,d,d])$, and $r(c,[d,d],[c,d,d])$ is compatible with $r(e,[f,g],[e,g,f])$, but $r(a,[b],[a,b])$ is not compatible with $r(e,[f,g],[e,g,f])$.

The following theorem is easily proved. It helps in the practical verification of admissibility and compatibility of atom sets.

Theorem 10-1: *Let $\mathcal{A}(r)$ be a non-empty set of atoms. Then:*

(1) *If $msg(\mathcal{A}(r))$ is admissible, then $\mathcal{A}(r)$ is admissible;*

(2) *$\mathcal{A}(r)$ is compatible iff $msg(\mathcal{A}(r))$ is admissible;*

(3) *$\mathcal{A}(r)$ is compatible iff $\mathcal{A}(r) \cup msg(\mathcal{A}(r))$ is compatible.*

Compatible sets $\mathcal{A}(r)$ of atoms are of great interest, because, by Theorem 10-1, $\mathcal{A}(r)$ and its msg are admissible, and the atoms of $\mathcal{A}(r)$ construct their parameters in the same way as their msg does. In other words, $\mathcal{A}(r)$ can be "collapsed" into a single atom, namely its msg, which "represents" the entire set $\mathcal{A}(r)$.

The following theorem trivially holds:

Theorem 10-2: *Let $\mathcal{A}(r)$ be a non-empty set of (ternary) atoms, and $r(m_1,m_2,m_3)$ be $msg(\mathcal{A}(r))$. Then:*

$$\mathtt{r(X_1,X_2,X_3)} \Leftrightarrow \mathtt{X_1{=}m_1} \wedge \mathtt{X_2{=}m_2} \wedge \mathtt{X_3{=}m_3}$$

is a logic algorithm that is complete wrt $\mathcal{A}(r)$, and correct wrt all the instances of $msg(\mathcal{A}(r))$.

But this theorem is independent of the admissibility and compatibility notions. So what are their roles? We are not interested in synthesizing a logic algorithm that covers "too many", if not all, examples beyond the given ones. Intuitively, admissibility and compatibility are thus meant to restrict the covered set, as captured in the following definition.

Definition 10-9: Let $\mathcal{A}(r)$ be a non-empty, admissible set of atoms, and $\mathcal{P}(\mathcal{A})$ a partition of $\mathcal{A}(r)$ into compatible subsets. The *natural extension* of $\mathcal{A}(r)$ wrt $\mathcal{P}(\mathcal{A})$ is the set of atoms that match the msg of some element of $\mathcal{P}(\mathcal{A})$.

The constraints on $\mathcal{R}$ can easily be generalized now so as to obtain a larger domain of applicability. Suppose that $\mathcal{R}$ is of arity $n = a+2b$, where $ab \neq 0$, and that its $(a+b+i)^{\text{th}}$ parameter is constructed from its $(a+i)^{\text{th}}$ parameter, for $1 \leq i \leq b$, while the first a parameters may or may not be used in these constructions. This generalization covers the approximation above in case $a=b=1$. Definition 10-5 can be changed as follows.

Definition 10-10: An atom $r(any_1,any_2,\ldots,any_a,\ in_1,in_2,\ldots,in_b,\ out_1,out_2,\ldots,out_b)$ is *admissible* wrt a and b iff the following two conditions hold:

$$\forall i \in [1,\ldots,b]\ \ cons(in_i) \subseteq cons(out_i)$$

$$\forall i \in [1,\ldots,b]\ \ vars(in_i) \subseteq vars(out_i).$$

The other definitions are unchanged, and both theorems still hold, if adapted.

10.3 The Ground Case

Let's start with the ground case, where all examples are ground atoms (as in Definition 6-1). The problem is stated in Section 10.3.1, and a method to solve it is explained in Section 10.3.2. Its correctness is discussed in Section 10.3.3, and it is illustrated in Section 10.3.4.

10.3.1 The Problem

Formally, the problem can first be posed as follows.
Given:

- a set $\mathcal{E}(r)$ of examples,

such that:

- $\mathcal{E}(r)$ is admissible,

find:

- a logic algorithm $LA(r)$,

such that:

- $LA(r)$ is defined in terms of the =/2 primitive only,
- $LA(r)$ is correct wrt some natural extension of $\mathcal{E}(r)$.

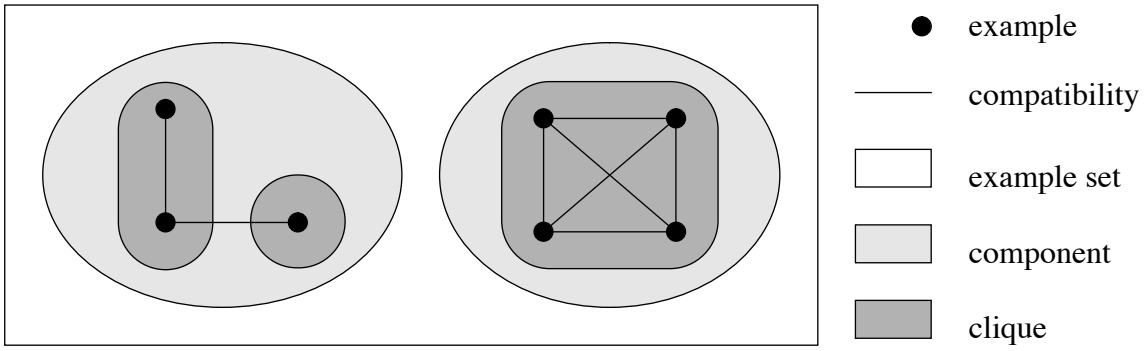
The initial problem is correctly solved iff the natural extension of $\mathcal{E}(r)$ is the unknown intended relation $\mathcal{R}$. This is of course impossible to verify, so the main issue is to infer a logic algorithm that gives maximal confidence that the initial problem is correctly solved.

10.3.2 A Method

It seems desirable to infer a logic algorithm that is correct wrt the largest possible natural extension of $\mathcal{E}(r)$. Indeed, a trivial solution would be to partition $\mathcal{E}(r)$ into singletons, and thus create a logic algorithm that is only correct wrt $\mathcal{E}(r)$. This may clearly not be what is intended, because $\mathcal{E}(r)$ is ground and finite. So the "less ground" the msgs of the subsets, the larger the natural extension. And the best way to obtain variables in the msgs of sets of ground atoms is to make these sets as large as possible. So the partition of $\mathcal{E}(r)$ should yield as few subsets as possible, provided they are all compatible. Because of the non-transitivity of the compatibility relation, this is a non-deterministic problem. It can be solved as follows.

First, decompose the graph of the compatibility relation over $\mathcal{E}(r)$ into its connected components. This is a classical graph theory problem, and is done by a depth-first traversal of the graph. The decomposition is known to be unique, and the algorithm is of complexity $O(m+q)$, where m is $\#\mathcal{E}(r)$, and q is the number of edges of the compatibility graph.

Second, partition each connected component into maximal compatible subgraphs (cliques). Non-deterministically retain, for each component, a partition that has the least number of cliques. So far, the algorithm can be graphically illustrated as follows:



Note that we could directly partition the entire $\mathcal{E}(r)$—rather than its connected components—into maximal compatible subgraphs. But the chosen approach is justifiable because non-compatible connected components are rather unusual in reality: the components obtained by the first step are often already cliques.

Finally, create a logic algorithm $LA(r)$ from the msgs of the obtained compatible subsets (suppose there are c of them):

$$
\begin{aligned}
&r(X_1,\ldots,X_n) \Leftrightarrow \\
&\quad X_1=m_{11} \wedge X_2=m_{12} \wedge \ldots \wedge X_n=m_{1n} \\
&\vee\ X_1=m_{21} \wedge X_2=m_{22} \wedge \ldots \wedge X_n=m_{2n} \\
&\vee\ \ldots \\
&\vee\ X_1=m_{c1} \wedge X_2=m_{c2} \wedge \ldots \wedge X_n=m_{cn}
\end{aligned}
$$

where $r(m_{i1},m_{i2},\ldots,m_{in})$ is the msg of component i ($1 \le i \le c$), and the X_k ($1 \le k \le n$)are different variables not occurring in the m_{ij} ($1 \le i \le c$, $1 \le j \le n$). The obtained logic algorithm can usually be rewritten more compactly, using equivalence-preserving transformation rules such as elimination of an $X=Y$ atom plus application of the substitution $\{Y/X\}$ to the corresponding disjunct, where Y is an existential variable.

The resulting MSG Method algorithm is thus as follows.

Algorithm 10-1: MSG Method (Ground Case).

(1) decompose the compatibility graph over $\mathcal{E}(r)$ into its connected components;
(2) partition each connected component into a least number of cliques;
(3) create a logic algorithm from the msgs of the obtained cliques.

Note that this algorithm actually also works for atom sets, by virtue of Definition 10-5 and Definition 10-7.

This algorithm is non-deterministic, finite, and never fails. Moreover, the synthesized logic algorithm is non-deterministic in general, finite, but may fail. For instance, considering Example 10-2, the atom *pcCompress(e,[e,2],X)* is covered by both disjuncts of the synthesized logic algorithm, namely via the answer substitutions $\{X/[e,3]\}$ and $\{X/[e,1,e,2]\}$.

The computation of the complexity of this algorithm goes as follows. Partitioning a graph into a maximum of k cliques (k being a given constant) is known to be an NP-complete problem [Garey and Johnson 79, page 193]. Hence the union of the first two steps already represents an NP-complete problem, as it requires an iteration over this problem, for $k=1\ldots m$.

10.3.3 Correctness

The following correctness theorem can now be established.

Theorem 10-3: *Let $\mathcal{E}(r)$ be a set of admissible (ground) examples, and LA(r) be the logic algorithm obtained by Algorithm 10-1, using a partition $\mathcal{P}(\mathcal{E})$ of $\mathcal{E}(r)$. Then LA(r) is defined in terms of the =/2 primitive only, and is correct wrt the natural extension of $\mathcal{E}(r)$ wrt $\mathcal{P}(\mathcal{E})$.*

Proof 10-3: Obvious, from Theorem 10-2, Definition 10-9, and Algorithm 10-1. ❑

10.3.4 Illustration

Let's illustrate the described method on some sample problems.

Example 10-6: Take the data of Example 10-2 again. The decomposition is $\{\{E_2,E_4,E_7,E_8\},\{E_3,E_5,E_6\}\}$. Both components are already compatible, so need not be split. Their msgs are *pcCompress(A,T,[A,1|T])* and *pcCompress(B,[B,s(N)|U],[B,s(s(N))|U])*. The created logic algorithm effectively is, after rewriting, the one given in Example 10-2.

Example 10-7: Let:

```
ℰ(pcFirstPlateau) = {
            pcFirstPlateau(b,[b],[],[b,b],[])          (E2)
            pcFirstPlateau(j,[j],[k],[j,j],[k])        (E6)
            pcFirstPlateau(m,[m,m],[],[m,m,m],[]) }    (E7)
```

and $a=1$, $b=2$. All examples are admissible. The decomposition yields one component, namely $\{E_2,E_6,E_7\}$, that is already compatible. The created logic algorithm is:

```
pcFirstPlateau(HL,TP,TS,P,S) ⇔
    HL=A ∧ TP=[A|U] ∧ TS=T ∧ P=[A,A|U] ∧ S=T
```

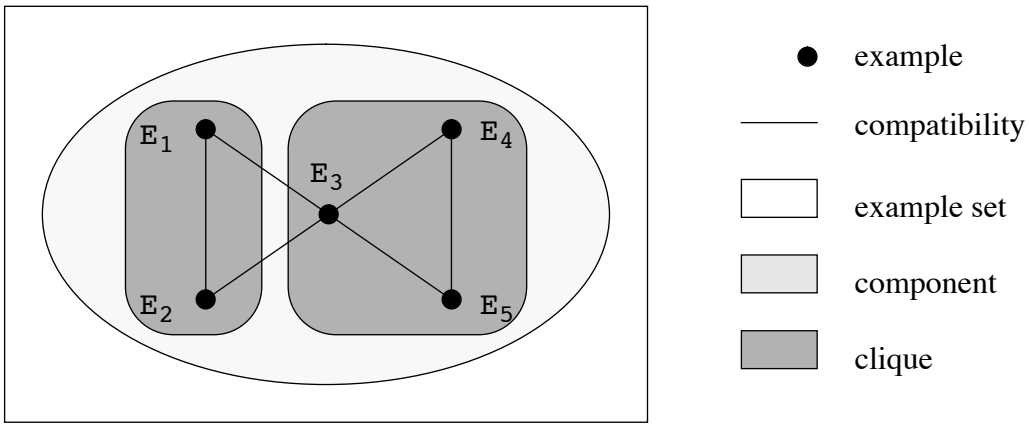
It may be rewritten as:

```
pcFirstPlateau(HL,TP,TS,P,S) ⇔
    P=[HL|TP] ∧ S=TS ∧ TP=[HL|_]
```
♦

Example 10-8: Let:

```
ℰ(foo) = {  foo(a,[],[a])                          (E1)
            foo(b,[c],[b,c])                       (E2)
            foo(d,[e,e],[d,e,e])                   (E3)
            foo(f,[g,h],[f,h,g])                   (E4)
            foo(i,[j,k,m],[i,k,j,m]) }             (E5)
```

and a=b=1. All examples are admissible. The compatibility graph has one connected component, namely $\mathcal{E}(foo)$ itself, which is not compatible. One of the two (symmetric) possible partitions into cliques is as follows:



Whatever the chosen clique partition, the created logic algorithm is:

```
foo(X,Y,Z) ⇔
    X=A ∧ Y=T ∧ Z=[A|T]
  ∨ X=A ∧ Y=[B,C|T]] ∧ Z=[A,C,B|T]
```
♦

10.4 The Non-Ground Case

The groundness assumption on $\mathcal{E}(r)$ is very strong. So we extend the MSG Method to sets $\mathcal{G}(r)$ of not necessarily ground examples, called general examples. The idea is to introduce alternatives in examples. Alternatives are naturally characterized by disjunctions and existential quantifiers.

Definition 10-11: A *general example* of predicate r/n is the existential closure of a disjunction of atoms of predicate r/n.

Note that a single-disjunct general example is different from an atomic property (a property with no body), because properties have universal quantifiers. If there is no ambiguity whether we are referring to an atomic property or a general example, we omit quantifiers.

For ease of notation, a general example $r(\boldsymbol{P},t_1,\boldsymbol{Q}) \vee r(\boldsymbol{P},t_2,\boldsymbol{Q}) \vee \ldots \vee r(\boldsymbol{P},t_n,\boldsymbol{Q})$ is denoted $r(\boldsymbol{P},\{t_1,t_2,\ldots,t_n\},\boldsymbol{Q})$, where t_1, t_2,…, t_n are terms, n>1, and $\boldsymbol{P}$, $\boldsymbol{Q}$ are possibly empty vectors of terms. We also allow a recursive use of this convention.

Example 10-9: The general example *append*([*X*],[*b*],[*a*,*b*]) represents the formula ∃*X* *append*([*X*],[*b*],[*a*,*b*]). The general example *append*({[*a*],[*b*]}, {[*c*],[*d*]}, [*a*,*d*]) represents the formula *append*([*a*],[*c*],[*a*,*d*]) ∨ *append*([*a*],[*d*],[*a*,*d*]) ∨ *append*([*b*],[*c*],[*a*,*d*]) ∨ *append*([*b*],[*d*],[*a*,*d*]).

Definition 10-12: Given a general example g, the set of *admissible alternatives* of g, denoted *adm(g)*, is defined as follows:

$$adm(g) = \{h \in \mathcal{A} \mid h \text{ is an instance of some disjunct of } g, \text{ and } h \text{ is admissible}\}.$$

Definition 10-13: Given a set $\mathcal{G}(r) = \{g_1, g_2, \ldots, g_m\}$ of general examples, the set of *admissible alternatives* of $\mathcal{G}(r)$, denoted *adm*($\mathcal{G}(r)$), is defined as follows:

$$adm(\mathcal{G}(r)) = \{ \{h_1, h_2, \ldots, h_m\} \mid h_i \in adm(g_i) \}.$$

Note that the admissible alternatives of a general example (set) are (sets of) atoms, not (sets of) ground examples. Also note that the *adm* relations may be infinite if variables occur in their arguments.

Example 10-10: Let:

```
G(pcPlateau)= { pcPlateau(1,[],A,[b],b)                (G1)
                pcPlateau(2,[B],B,[c,c],c)             (G2)
                pcPlateau(3,[C,C],C,[d,d,d],d) }       (G3)
```

and a=1, b=2. Then *adm*(G_2) contains *pcPlateau*(2,[*c*],*c*,[*c*,*c*],*c*), but not *pcPlateau*(2,[*b*], *b*,[*c*,*c*],*c*), nor *pcPlateau*(2,[*d*],*d*,[*c*,*c*],*c*). ♦

The new problem is stated in Section 10.4.1, and a method to solve it is in Section 10.4.2. Its correctness is discussed in Section 10.4.3, and it is illustrated in Section 10.4.4.

10.4.1 The Problem

Formally, the problem can now be expanded as follows:
Given:

- a set $\mathcal{G}(r)$ of general examples,

such that:

- $\mathcal{G}(r)$ has admissible alternatives,

find:

- a logic algorithm *LA(r)*,

such that:

- *LA(r)* is defined in terms of the =/2 primitive only,
- *LA(r)* is correct wrt some natural extension of some element of *adm*($\mathcal{G}(r)$).

10.4.2 A Method

The objective is again to produce partitions with the smallest possible number of subsets. This can be done as follows: Compute all possible admissible alternatives of $\mathcal{G}(r)$, and partition them using Algorithm 10-1. Select one of the solutions that have the least number of cliques. The resulting extended algorithm is thus as follows.

Algorithm 10-2: MSG Method (Non-ground Case).

(1) perform Algorithm 10-1 for every element in *adm*($\mathcal{G}(r)$), and select a logic algorithm resulting from the least number of cliques.

This algorithm is non-deterministic and NP-complete, as it is based on Algorithm 10-1. It always succeeds, but may be infinite. Moreover, the synthesized logic algorithm is non-deterministic in general, but finite, because Algorithm 10-1 produces such logic algorithms.

If a set *adm(g)* is infinite, then the search space must be reduced by applying some heuristics. Thus, given an atom $g = r(any,in,out)$, where $vars(<in,out>) \neq \emptyset$, we have to bridle the *adm* relation so that, in its search for a substitution σ for g, it enumerates a smaller space than the infinite $vars(g) \times \mathcal{T}$, and preferably a finite one. First, this search space can be pruned with type knowledge, such as:

- application-independent type knowledge, such as: *s*/1 is only applicable to integers;
- application-specific type knowledge, in case type information is given along with *g*.

Next, there are some particular cases where the search space is (or may be) reduced, taking advantage of the context:

- if $vars(out) = \emptyset$, then the range of σ is restricted to $\mathcal{U}(out)$; we may even first emit the *minimal hypothesis* that the range of σ is restricted to the finite set *cons(out)*;
- if $vars(in) = \emptyset$, then we may in a first approximation look for the least solution: the *strongly conservative hypothesis* restricts the range of σ to *cons(in)*; otherwise, the *weakly conservative hypothesis* restricts the range of σ to $\mathcal{U}(in)$.

Anyway, the resultant search space may be huge, so we have to be careful in practice to apply this method only when there are few variables in *g*, less than 3, say. There is no problem of course if the general example is ground, because *adm* then only needs to perform a finite enumeration.

10.4.3 Correctness

The following correctness theorem can easily be proven.

Theorem 10-4: *Let $\mathcal{G}(r)$ be a set of admissible general examples, and LA(r) be the logic algorithm obtained by Algorithm 10-2, using a partition $\mathcal{P}(\mathcal{G}')$ of some element $\mathcal{G}'(r)$ of $adm(\mathcal{G}(r))$. Then LA(r) is defined in terms of the =/2 primitive only, and is correct wrt the natural extension of $\mathcal{G}'(r)$ wrt $\mathcal{P}(\mathcal{G}')$.*

10.4.4 Illustration

Let's illustrate the described method on some sample problems. The first one involves existential variables, while the second involves disjunctions.

Example 10-11: Take the data of Example 10-10 again. If we restrict the functor set to {*b,c,d*}, the only element of $adm(\mathcal{G}(pcPlateau))$ is:

```
{ pcPlateau(1,[],b,[b],b)                    (G1')
  pcPlateau(2,[c],c,[c,c],c)                 (G2')
  pcPlateau(3,[d,d],d,[d,d,d],d) }           (G3')
```

The only clique is $\{G_1', G_2', G_3'\}$, and has *pcPlateau(s(N),T,A,*[*A*|*T*]*,A)* as msg. The created logic algorithm is, after rewriting, as follows:

```
pcPlateau(HN,TL,TX,L,X) ⇔
                    L=[X|TL] ∧ X=TX ∧ HN=s(N)  ♦
```

Example 10-12: Let:

```
G(pcPermutation) = {
            pcPermutation(a,[],[a])                    (G2)
            pcPermutation(b,[c],[b,c])                 (G3)
            pcPermutation(b,[c],[c,b])                 (G4)
            pcPermutation(d,{[e,f],[f,e]},[d,e,f])     (G5)
            pcPermutation(d,{[e,f],[f,e]},[d,f,e])     (G6)
            pcPermutation(d,{[e,f],[f,e]},[e,d,f])     (G7)
            pcPermutation(d,{[e,f],[f,e]},[e,f,d])     (G8)
            pcPermutation(d,{[e,f],[f,e]},[f,d,e])     (G9)
            pcPermutation(d,{[e,f],[f,e]},[f,e,d]) } (G10)
```

and a=b=1. There are 2^6=64 admissible alternatives of *G(pcPermutation)*: for general examples G_5 to G_{10}, we denote their first alternative by G_i', and their second alternative by G_i". There are two least partitions, namely {{G_2,G_3,G_5',G_6"}, {G_4,G_7',G_9"}, {G_8',G_{10}"}}, and {{G_2,G_3,G_5',G_6"}, {G_4,G_7',G_9"}, {G_8",G_{10}'}}. For the first partition, the created logic algorithm is as follows:

```
pcPermutation(HL,TP,P) ⇔
    HL=A ∧ TP=T ∧ P=[A|T]
  ∨ HL=A ∧ TP=[B|T] ∧ P=[B,A|T]
  ∨ HL=A ∧ TP=[B,C] ∧ P=[B,C,A]
```

For the second partition, the created logic algorithm is as follows:

```
pcPermutation(HL,TP,P) ⇔
    HL=A ∧ TP=T ∧ P=[A|T]
  ∨ HL=A ∧ TP=[B|T] ∧ P=[B,A|T]
  ∨ HL=A ∧ TP=[B,C] ∧ P=[C,B,A]                          ♦
```

10.5 Future Work

The MSG Method, as presented here, is sufficient for our future needs. But it may be enhanced along several lines.

First, as of now, the compatibility criterion of Definition 10-10 doesn't prescribe any restrictions on the first a parameters of an (a+2b)-ary atom. This holds because of the statement that they may or may not be involved in the construction of the last b parameters from the other b parameters. The current compatibility criterion is fine with a large number of applications. But it could be refined so as to include the following restriction: if (some of) the constants or variables of (some of) the first a parameters of atom s are used in the construction of (some of) the last b parameters of s, then the same must happen for atom t. This refinement should enlarge the number of successfully covered applications. Other refinements can be imagined, but the whole framework of correctness definitions and theorems for the MSG Method needs to be proved afresh each time.

Second, the notion of general examples could be extended to the notion of *constrained general examples*. Constraints, such as those generated by the *exp*/3 operator (Definition 7-25), could indeed be attached to general examples, and thus constrain the search for admissible alternatives of such constrained general examples.

Third, the decomposition of the arity n of predicate r into a+2b directly results from the applications we have in mind for the MSG Method. But such a restrictive setting may not be suitable for other needs. In order to generalize this setting, we could define the concept of mode, so that we may precisely describe which parameters are constructed in terms of which parameters.

Definition 10-14: Let r be an n-ary predicate symbol. A *mode* m_r for r is a total function from $\{1,2,\ldots,n\}$ into $\{in,in_1,in_2,\ldots,out,out_1,out_2,\ldots,any\}$. If $m_r(i)$ is X, then the i^{th} parameter of r is called an *X-parameter* of r. If $m_r(i)$ is *in or* in_j, then the i^{th} parameter of r is called an *input-parameter* of r. If $m_r(i)$ is *out or* out_j, then the i^{th} parameter of r is called an *output-parameter* of r.

A mode m_r is often written in the more suggestive form $r(m_r(1),m_r(2),\ldots,m_r(n))$. It should be noted that, despite the syntactic similarity, these "construction" modes have nothing to do with the "execution" modes, such as *r(ground,ground,any)*, that are often found in the logic programming literature. Indeed, "construction" modes are descriptions of static relationships between parameters, whereas "execution" modes are descriptions of dynamic states of parameters.

Example 10-13: The mode for *pcCreate*/5, as defined in Example 10-10, would be $pcCreate(in,in_1,in_2,out_1,out_2)$.

The intuition behind these "construction" modes is the following. The out_i-parameters must be constructed in terms of the in_i-parameters. The *in*-parameters may or may not be used in the construction of *output*-parameters. The *out*-parameters may or may not be constructed in terms of *input*-parameters. There are no constraints on *any*-parameters. Nothing prevents out_i-parameters to be constructed in terms of in_j-parameters, where $i \neq j$. Nothing prevents out_i-parameters to "invent" constants.

These modes are a possible starting point for a very refined version of the MSG Method, with enhanced admissibility and compatibility criteria.

10.6 Related Work

The related work for the MSG Method can be divided into two categories. First, there is the theoretical research on the concept of msg, and its practical applications. Second, there is research on techniques similar to the MSG Method.

As said earlier, the concepts of msg and anti-unification were introduced simultaneously, but independently, by [Plotkin 70, 71] and [Reynolds 70]. The original motivation is accredited to [Popplestone 70], who suggested that "since unification is useful in automatic deduction by the resolution method, its dual might prove helpful for induction". The main differences between both works are that Plotkin takes a logic-based approach and considers atomic formulas and clauses, whereas Reynolds is more concerned about an algebraic approach and only considers atomic formulas. But their results and msg algorithms are essentially the same.

These early works have spawned a lot of theoretical interest, among which the following studies of term lattices should be noted: [Huet 76] and [Lassez *et al*. 87].

The concepts of msg and anti-unification have naturally generated a lot of practical applications for the machine learning community, and especially from researchers on inductive logic programming. [Vere 75] infers conjunctive concept descriptions from positive and negative examples, using msgs. He introduces the useful notion of inverse substitution in order to formalize how the term lattice can be traversed "upwards". The problem of learning concept descriptions from negative examples is studied by [Lassez and Marriott 87]. Building on his semantic generalization model (surveyed in Section 7.5), [Buntine 88] suggests the notion of msg of two clauses relative some other clauses.

[Gegg-Harrison 89, 93] extends the computation of msgs to a second-order framework (and improves the complexity of the original algorithms) so as to infer logic program schemas for a logic programming tutor. Simultaneously, but independently, [Tinkham 90] also

uses a second-order lattice of logic program schemas so as to efficiently guide an MIS-like system for logic program synthesis from examples.

In terms of alternative techniques to the MSG Method, there is of course the whole literature on empirical learning, and inductive logic programming, as surveyed in Chapter 3. But it should be noted that the MSG Method only aims at the synthesis of a sub-class of concept descriptions, namely non-recursive algorithms that are implemented in terms of the =/2 primitive only, and whose intended relation, though unknown as a whole, is however known to feature a given construction pattern between its parameters. This extremely restrictive setting justifies the non-incremental approach, as well as the highly specialized MSG Method.

The technique of [Vere 75] is quite similar to ours in spirit, as the fundamental notion is the one of msg, and as no recursive concept descriptions are aimed at. But he only aims at conjunctive concept descriptions, whereas our technique is also able to infer disjunctive concept descriptions. Also, the MSG Method is based only on positive examples, and is only set in a first-order logic.

The notion of general example seems to be new. It is not related to noisy examples, because the existence of a correct admissible alternative of a general example is assumed.

10.7 Conclusion

Within a restricted setting, the MSG Method infers, from a finite set of general examples, a non-recursive logic algorithm that is defined in terms of the =/2 primitive only, and that is correct wrt a natural extension of the given examples. The underlying algorithm is non-deterministic.

If the resulting logic algorithm is judged, by whatever application-dependent heuristics, to be "not good enough", then the assumption that the examples can be covered by such a logic algorithm must be revised: recursion, or other predicates, or both, might be needed. This requires a more involved method. A possible solution is outlined in Section 14.2.4: it is called the *Synthesis Method*, because it automatically infers a property set of the intended relation (this is possible in some applications), and then uses the entire synthesis mechanism (as described in Part III) in order to infer a logic algorithm that covers the given examples and inferred properties.

So why not immediately use the Synthesis Method? The reason is that both methods tackle different classes of problems. The MSG Method and the Synthesis Method are a joint answer to the same problem: how to infer, from a finite set of general examples of an unknown relation that is however known to feature a given construction pattern between its parameters, a logic algorithm that is correct wrt a natural extension of the given examples. The MSG Method is the "base case" of the answer, because it doesn't look for recursion, and the Synthesis Method is the "structure case" of the answer, because it does look for recursion.

Another crucial question that pops up is: Why not use a "classical" concept learner? The reason for not doing so is that a concept learner, in its incremental approach and blind enumeration, doesn't take advantage of the knowledge about the construction pattern between the parameters of the intended relation. Concept learners are a reasonable approach if nothing is known about the intended relation. But this is not the case here, and more practical methods can then be developed.

Note that the MSG Method is part of our tool-box for solving sub-problems occurring during synthesis, and not a solution to the overall problem of synthesis from specifications by examples (and properties).

III A LOGIC ALGORITHM SYNTHESIS MECHANISM

Synthesis, in wider philosophical use and generally:
the putting together of parts or elements so as to make a complex whole;
the combination of immaterial or abstract things, or of elements
into an ideal or abstract whole (opposed to analysis).

—The Oxford English Dictionary

In this third part, we present a particular logic algorithm synthesis mechanism, designed from the building blocks provided in Part II. Thus, in Chapter 11, we give an intuitive overview of the entire synthesis mechanism, so as to give the reader the feel for its working. Chapter 12 gives full detail about the Expansion Phase of synthesis, that is the first four steps, while Chapter 13 does similarly with the Reduction Phase, that is the remaining three steps. Finally, in Chapter 14, we evaluate the resulting mechanism.

11 Overview of the Synthesis Mechanism

In this chapter, we give an intuitive overview of the entire synthesis mechanism, so as to give the reader the feel for its working. Thus, in Section 11.1, we first motivate the desired features of this mechanism. Then, in Section 11.2, we argue for a series of preliminary restrictions of this mechanism, so as to keep the presentation simple until the discussion of its extensions (see Chapter 14). Finally, in Section 11.3, we perform a sample synthesis.

11.1 Desired Features

Considering the state of the art of inductive synthesis from examples (as seen in Chapter 3), we now decide on, and motivate, some desired features of the synthesis mechanism. These decisions are about the specification language, the logic algorithm language, the degree of automation of synthesis, the types of performed reasoning, and the features of the chosen synthesis strategy. In the sequel, "synthesis system" stands for an implementation of the synthesis mechanism.

Specification Language

We gave a very general definition of specifications by examples and properties in Chapter 6, but postponed the actual decisions about the concrete languages for examples and properties to this Part III, in order to be independent of such decisions for as long as possible. During the course of Part II, various languages have been proposed, according to how much could actually be handled for each notion or method we introduced. Basically, the property language is each time some extension or restriction to Horn clauses, the differences lying in whether negation and recursion are allowed or not. And the example languages differ in whether negative examples are required or not. In summary:

- in Chapter 6 (specification approach), a sample language is proposed for illustration purposes only;
- in Chapter 7 (synthesis framework), all definitions are language-independent;
- in Chapter 8 (algorithm schemas), there is no concern about specifications;
- in Chapter 9 (Proofs-as-Programs Method), the example language is irrelevant, but properties are constrained to be (possibly recursive) Horn clauses (without negation);
- in Chapter 10 (MSG Method), the property language is irrelevant, whereas only positive examples are taken into account, be they ground or general examples.

So we could here simply choose the largest common subset of all the proposed relevant languages. But two additional restrictions can still be made: Are negative examples required? Are recursive properties allowed? Negative examples are usually expected in order to avoid over-generalization. But it can be argued (see Section 14.2.1) that negative information is not required here. Then, although we could handle recursive Horn clauses, we limit properties to be non-recursive, so as to focus on synthesis from incomplete specifications. Recursion discovery is indeed a major challenge of algorithm synthesis, and a challenge that we want to pick up. So there is no need to give that information away by means of recursive properties. The refined definition of specifications by examples and properties is thus as follows:

Definition 11-1: A *specification by examples and properties* of a procedure for predicate r/n, denoted $EP(r)$, consists of:

- a set $\mathcal{E}^+(r)$ of positive examples of r (that is, ground atoms whose n-tuples are supposed to belong to the intended relation $\mathcal{R}$); for simplicity, we denote $\mathcal{E}^+(r)$ by $\mathcal{E}(r)$;
- a set $\mathcal{P}(r)$ of properties of r (that is, non-recursive Horn clauses whose heads are atoms of predicate r).

We thus require multiple, ground, single-predicate, relational, positive-only, pre-synthesis examples that are chosen in a consistent way by a human specifier who knows the intended relation. The predicates used in the properties constitute a partial basis set, and thus a partial conceptual bias for synthesis. For convenience, and as there is no ambiguity in doing so, we drop the universal quantifications of properties. Overall, the specification language is quite expressive and readable. Specifications by examples and properties are usually incomplete, and hence ambiguous, but minimal. There is a danger of internal inconsistency and redundancy in such specifications, though. The synthesis mechanism should be robust to the ordering and choice of examples and properties.

Logic Algorithm Language

We gave a very general definition of logic algorithms in Chapter 4, but postponed the actual decisions about the concrete language for bodies to this Part III, in order to be independent of such decisions for as long as possible. During the course of Part II, various languages have been proposed, according to how much could actually be handled for each notion or method we introduced. In summary:

- in Chapter 6 (specification approach), there is no concern about logic algorithms;
- in Chapter 7 (synthesis framework), logic algorithms are required to be implemented in terms of primitives only; for more specific purposes (see Section 7.2.2 and Section 7.3.2), bodies of logic algorithms are constrained to be in disjunctive normal form (with negation, that is);
- in Chapter 8 (algorithm schemas), all definitions are language-independent;
- in Chapter 9 (Proofs-as-Programs Method), bodies of logic algorithms are constrained to be in prenex disjunctive normal form, though without negation;
- in Chapter 10 (MSG Method), the imposed restrictions are only on target logic algorithms, and hence irrelevant.

So we here simply choose the largest common subset of all the proposed relevant languages. The refined definition of logic algorithms is thus as follows:

Definition 11-2: A *logic algorithm* defining a predicate r/n, denoted $LA(r)$, is a closed well-formed formula of the form:

$$\forall X_1 \ldots \forall X_n \quad r(X_1,\ldots,X_n) \Leftrightarrow F$$

where the X_i are distinct variables, and F is a well-formed formula in prenex disjunctive normal form, without negation. The atom $r(X_1,\ldots,X_n)$ is called the *head*, and F is called the *body* of the logic algorithm.

This definition actually constitutes a partial syntactic bias for the synthesis. As usual, we drop the universal quantifications in front of the heads, as well as any existential quantifications at the beginnings of bodies of logic algorithms. Moreover, we often write logic algorithms in a more compact form, using one of De Morgan's laws in order to merge disjuncts.

Degree of Automation

With incomplete specifications, it is more realistic to strive for an interactive synthesis mechanism, so as to explicitly disambiguate some situations, rather than have a default iteration over (all) possible choices. The initiative should be on the system's side, because the latter would otherwise slide from the synthesizer category into the design assistant category. The questions asked to the specifier should be kept simple, the universe of the dialogue being restricted to the specified predicate and the primitive predicates. In other words, predicates that the mechanism invents during the synthesis should not occur in any dialogue, as the specifier is not familiar with them. Also, the language of questions and answers should not be more

"complicated" than the actual specification language. This means that the answers could actually be seen as extensions to the initially given specification, so that if the extended specification had been given right away, the synthesis would have been fully automatic. In our case, questions to the specifier could be of the following kinds:

- *classification queries*, such as "is a given ground atomic formula *A* a positive example of the intended relation $\mathcal{R}$?"; a correct answer is either *yes* (iff the tuple of terms extracted from *A* belongs to $\mathcal{R}$) or *no* (otherwise);
- *existential queries*, such as "is a given atomic formula *A* satisfiable in the intended model $\Im$?"; a correct answer is either <*yes*, σ> (iff $A\sigma$ is valid in $\Im$, where σ is a substitution) or *no* (otherwise);
- *universal queries*, such as "is a given atomic formula *A* valid in the intended model $\Im$?"; a correct answer is either *yes* (iff *A* is valid in $\Im$) or *no* (otherwise);
- *conditional queries*, such as "under what (conjunctive) condition is a given atomic formula *A* valid in $\Im$?"; a correct answer is either <*yes, C*> (iff $A \Leftarrow C$ is valid in $\Im$, where *C* is a conjunction of literals) or *never* (otherwise).

Answers such as "*I don't know*" should also be taken into account. Classification queries are a particular case of universal queries and of existential queries, namely when there are no variables in *A*. Universal queries are a particular case of conditional queries, namely when *C* reduces to *true*. Similarly to our assumption about the consistency of specifications with intended relations, we assume that the specifier's answers to queries are noise-free. Note that we talk about questions to a (human) specifier here, and not about the more general problem of asking questions to some oracle. At this level of the discourse, we are not interested in knowing whether a mechanized oracle is queried during synthesis or not.

There is another level of specifier interference during synthesis. Indeed, there is an infinity of algorithms defining a given intended (and computable) relation. For instance, if we only consider divide-and-conquer algorithms, there are design choices to be made about the induction parameter, its decomposition, a well-founded relation over its domain, and so on. A synthesis mechanism should then be able to discover a significant sub-family of algorithms within its search space, one algorithm per combination of taken choices. Given a specification of the *sort*/2 predicate, one would thus expect *Insertion-Sort*, *Merge-Sort*, *Quick-Sort*, and so on [Clark and Darlington 80] [Lau 89], and not just one of these algorithms. While the derivability of a large family of algorithms is an impressive (and desired) feature, this may go counter the specifier's expectations, because s/he only needs one algorithm, or because s/he has some understanding of the synthesis mechanism and wants to experiment with some very specific design choices. A synthesis system should thus accept hints about preferences from the specifier, as well as "think aloud" so that the specifier may interfere.

Finally, given some synthesized logic algorithm(s), how does the specifier continue to interact with the synthesis system? S/he could experiment with the algorithm by executing a logic program derived [Deville 90] from it, or s/he could directly analyze the algorithm itself (this calls for the invention of meaningful names of system-generated variables and predicates). In either case, an error might be discovered, or maybe even the requirements change. Traditional debugging techniques can then be invoked at the logic algorithm or logic program level. Ideally, though, the original specification is debugged to reflect the corrected/updated requirements, and a new synthesis is performed.

Kinds of Inference

With specifications by examples (which traditionally give rise to inductive synthesis) and properties (which, as a particular case of axioms, traditionally give rise to deductive synthesis), it should be natural to use both inductive and deductive inference in the synthesis mech-

anism, whichever is best suited for each sub-task. Other, obviously less well-founded decisions, would be (1) to do purely inductive synthesis only from the examples, and to use the properties only as integrity constraints in order to reject candidate logic algorithms, or (2) to do purely deductive/constructive synthesis only from the properties, and to use the examples only as test data in order to reject candidate logic algorithms. Moreover, other kinds of inference (such as analogical inference, or abductive inference) could be used as well.

Strategy Criteria

Stepwise synthesis is argued for in Section 7.3 because it allows different methods to be deployed at each step (rather than having a unique method handle a wide variety of different tasks), and because it yields monitoring points where correctness and comparison criteria can be applied (rather than having a unique monitoring point).

Schema-guided synthesis is argued for in Section 8.1 because schemas are an interesting way to incorporate algorithm design knowledge into a synthesis process. Schema-guided synthesis is naturally a stepwise synthesis, as the predicate variables are not all instantiated at the same time. A most interesting approach is then advocated in Section 8.3, namely to deploy an entire tool-box of predicate variable instantiating methods, rather than a unique method. In Chapter 9 (Proofs-as-Programs Method) and Chapter 10 (MSG Method), we describe two of the more sophisticated methods we have developed so far. Note that these methods are entirely dissociated from specific schemas or predicate variables.

The divide-and-conquer strategy is presented as an attractive strategy in Section 8.2, because of its diverse applicability, simplicity of application, efficient results, and suitability with incomplete specifications. The here chosen approach for developing a synthesis mechanism is thus stepwise synthesis guided by a divide-and-conquer schema.

As a first approximation to this approach, this thesis develops a synthesis mechanism that reflects a hardwired, fixed sequence of instantiations of the predicate variables of (some version of) the divide-and-conquer schema, as well as hardwired, fixed mapping between these predicate variables and the methods of the tool-box. The chosen sequence of steps is presented in Example 8-10; the chosen mapping between steps and methods is presented in the subsequent chapters, as other methods are introduced. Note that this approach also amounts to hardwiring the divide-and-conquer schema into the synthesis mechanism. The development of a synthesis mechanism that is parameterized on schemas, and that has no fixed sequences of steps or mappings between steps and methods, is considered future research.

Now, what is the chosen strategy for stepwise synthesis? We consider that all the specification information should be available for every inference during synthesis, and that a single run through all synthesis steps is hence sufficient. We thus favor a non-incremental (all-at-once) presentation of examples and properties to the synthesis mechanism. This leads to non-incremental synthesis, which is more "disciplined" than incremental synthesis, where parts of algorithms are continuously designed/debugged/rejected from partial information (extracted from an incomplete specification). The drawback of our approach is that we can't use the nice learnability-in-the-limit results that are known for incremental synthesis.

Note the difference between incremental synthesis and the incremental usage of a non-incremental synthesis mechanism: the former continuously debugs a unique algorithm from an increasingly large specification, whereas the latter amounts to presenting increasingly large specifications to a synthesis mechanism that always starts from scratch and generates as many different algorithms. The latter action only makes sense when a specification turns out to be not specific enough. But this amounts to specification augmentation, not to algorithm debugging. A useful generalization of specification augmentation is full-blown specification debugging.

So we adopt the non-incremental synthesis strategy presented in Section 7.3.2, which reflects monotonically decreasing synthesis (and is actually also monotonically increasing, if one also considers the expanded logic algorithms), as well as consistent synthesis. We should also strive to make each synthesis step comply with Theorem 7-7.

11.2 Preliminary Restrictions

In order to keep the presentation of the synthesis mechanism simple, we now decide on three preliminary restrictions.

First, regarding the schema underlying the mechanism, we retain version 3 of the divide-and-conquer schema for the theoretical presentation:

```
R(X,Y) ⇔
            Minimal(X)      ∧ Solve(X,Y)
   ∨ ∨1≤k≤c NonMinimal(X)   ∧ Decompose(X,HX,TX)
                            ∧ Discriminatek(HX,TX,Y)
                            ∧ (     SolveNonMink(HX,TX,Y)
                                  |
                                    R(TX,TY)
                              ∧ Processk(HX,HY)
                              ∧ Composek(HY,TY,Y)       )
```

This amounts to the support of binary relations, induction on a single parameter, a single minimal form, a single non-minimal form, and non-recursive non-minimal cases.

The support of version 4 (relations of any non-zero arity) is actually a pretty straightforward extension, but the needed vectorization implies a useless syntactic complication of notations. Some sample illustrations during the presentation of the synthesis mechanism actually require version 4, but we assume the reader can easily extrapolate how the theory has to be expanded in order to accommodate these illustrations. Version 4 is the schema that is actually supported by the implementation of the mechanism. The support of version 5 (any number of minimal or non-minimal forms) and version 6 (compound induction parameters) is considered future research. Note that Section 5.2.3 shows that single minimal forms and non-minimal forms are more general than one might believe at first sight.

Second, the mechanism should support any inductively defined data-types (such as integers, lists, bags, sets, trees, graphs, grammars, and so on) for potential induction parameters, provided their definitions have been made known to it. But whenever concrete data-types need to be discussed, we arbitrarily restrict the support to lists and integers.

Third, we only aim at the synthesis of single-loop logic algorithms. In other words, we assume that the only "loop" is the one that is achieved in the schema by the recursion on the induction parameter, and that none of the instances of the predicate variables is defined recursively (possibly as a divide-and-conquer logic algorithm).

11.3 A Sample Synthesis

Instantiating some predicate variable(s) of the schema above is a synthesis step. The synthesis mechanism is expressed as the following sequence of steps (see Example 8-10):

- Step 1: Syntactic creation of a first approximation;
- Step 2: Synthesis of *Minimal* and *NonMinimal*;
- Step 3: Synthesis of *Decompose*;
- Step 4: Syntactic introduction of the recursive atoms;
- Step 5: Synthesis of *Solve* and the $SolveNonMin_k$;

- Step 6: Synthesis of the $Process_k$ and $Compose_k$;
- Step 7: Synthesis of the $Discriminate_k$.

The chosen, fixed mapping between the synthesis steps and the methods of our tool-box is explained in the next two chapters, where other methods are introduced.

Example 11-1: Let's now perform an intuitive overview of these seven steps and illustrate them on the *compress*/2 relation. Before proceeding to the detailed explanations in the next two chapters, the reader is thus assumed to perform some learning-from-traces while studying this sample synthesis. Reconsider *EP(compress)* as given in Example 6-1:

```
ℰ(compress) = { compress([],[])                              (E1)
                compress([a],[a,1])                          (E2)
                compress([b,b],[b,2])                        (E3)
                compress([c,d],[c,1,d,1])                    (E4)
                compress([e,e,e],[e,3])                      (E5)
                compress([f,f,g],[f,2,g,1])                  (E6)
                compress([h,i,i],[h,1,i,2])                  (E7)
                compress([j,k,m],[j,1,k,1,m,1]) }            (E8)
𝒫(compress) = { compress([X],[X,1])                          (P1)
                compress([X,Y],[X,2]) ⇐ X=Y                  (P2)
                compress([X,Y],[X,1,Y,1]) ⇐ X≠Y }            (P3)
```

The chosen stepwise strategy is such that the synthesized logic algorithms progress downwards, while their expansions progress upwards. The synthesis proceeds as follows.

Step 1: *Syntactic Creation of a First Approximation*

Step 1 creates $LA_1(r)$ by setting its body to *true*. For instance, $LA_1(compress)$ is as follows:

```
compress(L,C) ⇔
    true                                              {E1,...,E8}
```

The set annotation to a disjunct explains which examples are covered by that disjunct. This logic algorithm always holds, and thus covers all given examples.

The expanded version of $LA_1(compress)$, that is $exp(LA_1(compress))$, is as follows:

```
compress(L,C) ⇔
    true      ∧     L=[] ∧ C=[]                           {E1}
                ∨   L=[a] ∧ C=[a,1]                       {E2}
                ∨   L=[b,b] ∧ C=[b,2]                     {E3}
                ∨   L=[c,d] ∧ C=[c,1,d,1]                 {E4}
                ∨   L=[e,e,e] ∧ C=[e,3]                   {E5}
                ∨   L=[f,f,g] ∧ C=[f,2,g,1]               {E6}
                ∨   L=[h,i,i] ∧ C=[h,1,i,2]               {E7}
                ∨   L=[j,k,m] ∧ C=[j,1,k,1,m,1]           {E8}
```

Note that this expanded version is a mere syntactic translation of the example set.

In the sequel, we only compute expansions that involve the variables that correspond to parameters in the schema. Also note that the inferences of each Step i (where $i \in \{2,\ldots,7\}$) are actually based on $exp(LA_{i-1}(r))$.

Step 2: *Synthesis of Minimal and NonMinimal*

An instantiation of the *Minimal* (respectively *NonMinimal*) predicate variable tests whether the induction parameter is of a minimal (respectively non-minimal) form. These forms must be mutually exclusive over the domain of the induction parameter. Step 2 yields $LA_2(r)$ by

first selecting an induction parameter, and then instantiating the *Minimal* and *NonMinimal* predicate variables by means of a so-called *Database Method*, which here relies on a database of type-specific form-identifying predicates.

For instance, we assume that Step 2 selects *L* as the induction parameter, identifies its type as being a list, and selects the empty list as the (unique) minimal form (which is identified by the atomic formula *L*=[]), and the non-empty list as the (unique) non-minimal form (which is identified[17] by the atomic formula *L*=[_|_]). *LA*$_2$*(compress)* is thus as follows:

```
compress(L,C) ⇔
    L=[]                                              {E1}
  ∨ L=[_|_]                                         {E2,...,E8}
```

Note the usage of anonymous variables: we are here merely interested in testing of what form the induction parameter is, not in knowing how it fits this test. This logic algorithm expresses that *compress(L,C)* holds whenever *L* is either an empty list or a non-empty list. This is strictly less often than always, as *L* now has to be a list (or at least a pseudo-list).

The expanded version of *LA*$_2$*(compress)*, that is *exp(LA*$_2$*(compress))*, is as follows:

```
compress(L,C) ⇔
    L=[]       ∧      L=[] ∧ C=[]                     {E1}
  ∨ L=[_|_]    ∧      L=[a] ∧ C=[a,1]                 {E2}
                  ∨   L=[b,b] ∧ C=[b,2]               {E3}
                  ∨   L=[c,d] ∧ C=[c,1,d,1]           {E4}
                  ∨   L=[e,e,e] ∧ C=[e,3]             {E5}
                  ∨   L=[f,f,g] ∧ C=[f,2,g,1]         {E6}
                  ∨   L=[h,i,i] ∧ C=[h,1,i,2]         {E7}
                  ∨   L=[j,k,m] ∧ C=[j,1,k,1,m,1]     {E8}
```

Note the redundancy between the two *L*=[] atoms in the minimal case: one of them is a synthesized atom, whereas the other one is a trailing atom introduced by expansion.

Step 3*: Synthesis of Decompose*

An instantiation of the *Decompose* predicate variable deterministically decomposes, in the non-minimal case, the induction parameter, say *X*, into a vector ***HX*** of heads and a vector ***TX*** of tails, the tails *TX*$_i$ being smaller than *X* according to some well-founded relation. These tails are meant for the recursive computation of the tails *TY*$_i$ of the other parameter, say *Y*. Step 3 yields *LA*$_3$*(r)* by instantiating the *Decompose* predicate variable by means of the *Database Method*, which here relies on a database of type-specific decomposition predicates.

For instance, we assume that Step 3 selects a simple head-tail decomposition of the induction parameter *L*. This is performed by the atomic formula *L*=[*HL*|*TL*]. *LA*$_3$*(compress)* is thus as follows:

```
compress(L,C) ⇔
    L=[]                                              {E1}
  ∨ L=[_|_]    ∧ L=[HL|TL]                        {E2,...,E8}
```

Note the coincidental redundancy between the instantiations of the *NonMinimal* and *Decompose* predicate variables. This logic algorithm is thus equivalent to its predecessor.

The expanded version of *LA*$_3$*(compress)*, that is *exp(LA*$_3$*(compress))*, is as follows:

17. Pseudo-lists, such as [*a*|*b*], are actually also identified by this formula.

```
compress(L,C) ⇔
    L=[]       ∧     L=[] ∧ C=[]                          {E1}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧     L=[a] ∧ C=[a,1]
                   ∧ HL=a ∧ TL=[]                         {E2}
                 ∨   L=[b,b] ∧ C=[b,2]
                   ∧ HL=b ∧ TL=[b]                        {E3}
                 ∨   L=[c,d] ∧ C=[c,1,d,1]
                   ∧ HL=c ∧ TL=[d]                        {E4}
                 ∨   L=[e,e,e] ∧ C=[e,3]
                   ∧ HL=e ∧ TL=[e,e]                      {E5}
                 ∨   L=[f,f,g] ∧ C=[f,2,g,1]
                   ∧ HL=f ∧ TL=[f,g]                      {E6}
                 ∨   L=[h,i,i] ∧ C=[h,1,i,2]
                   ∧ HL=h ∧ TL=[i,i]                      {E7}
                 ∨   L=[j,k,m] ∧ C=[j,1,k,1,m,1]
                   ∧ HL=j ∧ TL=[k,m]                      {E8}
```

Step* 4: *Syntactic Introduction of the Recursive Atoms

Now that ***TX*** is defined, we can introduce a conjunction of recursive atoms, at the rate of one per TX_i. This introduces a vector ***TY*** of tails of *Y*, at the rate of one per TX_i.

For instance, LA_4*(compress)* is as follows:

```
compress(L,C) ⇔
    L=[]                                                  {E1}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ compress(TL,TC)                  {E2,...,E8}
```

This logic algorithm expresses that *compress(L,C)* holds iff *L* is either an empty list or a non-empty list whose tail has a compression. As this compression isn't really computed yet, this only amounts to checking whether *L* is a real list (and this hence eliminates pseudo-lists). The expanded version of LA_4*(compress)*, that is *exp*(LA_4*(compress))*, is as follows:

```
compress(L,C) ⇔
    L=[]       ∧     L=[] ∧ C=[]                          {E1}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ compress(TL,TC)
               ∧     L=[a] ∧ C=[a,1]
                   ∧ HL=a ∧ TL=[] ∧ TC=[]                 {E2}
                 ∨   L=[b,b] ∧ C=[b,2]
                   ∧ HL=b ∧ TL=[b] ∧ TC=[b,1]             {E3}
                 ∨   L=[c,d] ∧ C=[c,1,d,1]
                   ∧ HL=c ∧ TL=[d] ∧ TC=[d,1]             {E4}
                 ∨   L=[e,e,e] ∧ C=[e,3]
                   ∧ HL=e ∧ TL=[e,e] ∧ TC=[e,2]           {E5}
                 ∨   L=[f,f,g] ∧ C=[f,2,g,1]
                   ∧ HL=f ∧ TL=[f,g] ∧ TC=[f,1,g,1]       {E6}
                 ∨   L=[h,i,i] ∧ C=[h,1,i,2]
                   ∧ HL=h ∧ TL=[i,i] ∧ TC=[i,2]           {E7}
                 ∨   L=[j,k,m] ∧ C=[j,1,k,1,m,1]
                   ∧ HL=j ∧ TL=[k,m] ∧ TC=[k,1,m,1]       {E8}
```

Step 5: *Synthesis of Solve and the SolveNonMin$_k$*

An instantiation of the *Solve* (respectively *SolveNonMin$_k$*) predicate variable computes, in the minimal case (respectively the non-recursive, non-minimal case), the value of the other parameter *Y* from the induction parameter *X*. Step 5 yields *LA$_5$(r)* by using similar methods to those of Steps 6 and 7.

For instance, in case *L* is the empty list, its compression *C* is the empty list as well. This is performed by the atomic formula *C*=[]. There is no non-recursive, non-minimal case, and hence no need to instantiate some *SolveNonMin$_k$*. *LA$_5$(compress)* is thus as follows:

```
compress(L,C) ⇔
    L=[]       ∧ C=L                                  {E1}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ compress(TL,TC)                {E2,...,E8}
```

This logic algorithm holds strictly less often than its predecessor, as the minimal case is now correctly computed.

The expanded version of *LA$_5$(compress)*, that is *exp(LA$_5$(compress))*, is as follows.

```
compress(L,C) ⇔
    L=[]       ∧ C=L
               ∧     L=[] ∧ C=[]                      {E1}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ compress(TL,TC)
               ∧     L=[a] ∧ C=[a,1]
                   ∧ HL=a ∧ TL=[] ∧ TC=[]             {E2}
                 ∨   L=[b,b] ∧ C=[b,2]
                   ∧ HL=b ∧ TL=[b] ∧ TC=[b,1]         {E3}
                 ∨   L=[c,d] ∧ C=[c,1,d,1]
                   ∧ HL=c ∧ TL=[d] ∧ TC=[d,1]         {E4}
                 ∨   L=[e,e,e] ∧ C=[e,3]
                   ∧ HL=e ∧ TL=[e,e] ∧ TC=[e,2]       {E5}
                 ∨   L=[f,f,g] ∧ C=[f,2,g,1]
                   ∧ HL=f ∧ TL=[f,g] ∧ TC=[f,1,g,1]   {E6}
                 ∨   L=[h,i,i] ∧ C=[h,1,i,2]
                   ∧ HL=h ∧ TL=[i,i] ∧ TC=[i,2]       {E7}
                 ∨   L=[j,k,m] ∧ C=[j,1,k,1,m,1]
                   ∧ HL=j ∧ TL=[k,m] ∧ TC=[k,1,m,1]   {E8}
```

Note the redundancy between the two *C*=[] atoms in the minimal case: one of them is a synthesized atom, whereas the other one is a trailing atom introduced by expansion.

Step 6: *Synthesis of the Process$_k$ and Compose$_k$*

An instantiation of a *Process$_k$* predicate variable transforms the heads ***HX*** of the induction parameter *X* into a vector ***HY*** of heads of the other parameter *Y*. An instantiation of a *Compose$_k$* predicate variable computes the other parameter *Y* from its heads ***HY*** (obtained by processing the ***HX***) and tails ***TY*** (obtained by recursion on the ***TX***). Step 6 does this simultaneously, and hence actually looks for an instantiation of a *ProcComp$_k$* predicate variable that computes the other parameter *Y* from the heads ***HX*** and tails ***TY***. Step 6 yields *LA$_6$(r)* by first invoking the *MSG Method* (see Chapter 10) and assessing its results via some heuristics: if these results are not satisfying, then a so-called *Synthesis Method* is invoked.

For instance, in our case the MSG Method is sufficient: it discovers 2 different ways of computing *C* from *HL* and *TC*, so the schema variable *c* of sub-cases is instantiated with 2.

This is actually shown in full detail in Example 10-2 and Example 10-6, where the following instance of *ProcComp* is synthesized:

```
pcCompress(HL,TC,C) ⇔
    C=[HL,1|TC]
  v C=[HL,s(s(N))|TTC] ∧ TC=[HL,s(N)|TTC]
```

LA_6*(compress)* is thus as follows:

```
compress(L,C) ⇔
    L=[]       ∧ C=L                                          {E1}
  v L=[_|_]    ∧ L=[HL|TL]
               ∧ compress(TL,TC)
               ∧ C=[HL,1|TC]                        {E2,E4,E7,E8}
  v L=[_|_]    ∧ L=[HL|TL]
               ∧ compress(TL,TC)
               ∧ C=[HL,s(s(N))|TTC] ∧ TC=[HL,s(N)|TTC]
                                                    {E3,E5,E6}
```

Note how the splitting of the non-minimal case into 2 sub-cases affects a partitioning of the examples that were covered by the original case. Basically, the added atoms express that the first value of the compression *C* is necessarily *HL*, and that the first counter of *C* is either 1 (in which case the tail of *C* is simply *TC*), or an integer *s(s(N))* strictly greater than 1 (in which case the tail of *C* is the tail of *TC*, the first value of *TC* is also *HL*, and the first counter of *TC* is *s(N)*).

The expanded version of LA_6*(compress)*, that is *exp*(LA_6*(compress))*, is as follows:

```
compress(L,C) ⇔
    L=[]       ∧ C=L
               ∧     L=[] ∧ C=[]                            {E1}
  v L=[_|_]    ∧ L=[HL|TL]
               ∧ compress(TL,TC)
               ∧ C=[HL,1|TC]
               ∧     L=[a] ∧ C=[a,1]
                   ∧ HL=a ∧ TL=[] ∧ TC=[]                   {E2}
                 v   L=[c,d] ∧ C=[c,1,d,1]
                   ∧ HL=c ∧ TL=[d] ∧ TC=[d,1]               {E4}
                 v   L=[h,i,i] ∧ C=[h,1,i,2]
                   ∧ HL=h ∧ TL=[i,i] ∧ TC=[i,2]             {E7}
                 v   L=[j,k,m] ∧ C=[j,1,k,1,m,1]
                   ∧ HL=j ∧ TL=[k,m] ∧ TC=[k,1,m,1]         {E8}
  v L=[_|_]    ∧ L=[HL|TL]
               ∧ compress(TL,TC)
               ∧ C=[HL,s(s(N))|TTC] ∧ TC=[HL,s(N)|TTC]
               ∧     L=[b,b] ∧ C=[b,2]
                   ∧ HL=b ∧ TL=[b] ∧ TC=[b,1]               {E3}
                 v   L=[e,e,e] ∧ C=[e,3]
                   ∧ HL=e ∧ TL=[e,e] ∧ TC=[e,2]             {E5}
                 v   L=[f,f,g] ∧ C=[f,2,g,1]
                   ∧ HL=f ∧ TL=[f,g] ∧ TC=[f,1,g,1]         {E6}
```

Note that the internal variables *N* and *TTC* do not appear in the expansions.

Step 7: *Synthesis of the Discriminate*$_k$

An instantiation of the *Discriminate*$_k$ predicate variable tests the values of ***HX***, ***TX***, and *Y* in order to see whether *ProcComp*$_k$ is applicable. Step 7 yields *LA*$_7$*(r)* by invoking the *Proofs-as-Programs Method* (see Chapter 9) and generalizing its results via some heuristics.

For instance, in our case the Proofs-as-Programs Method performs exactly as shown in Example 9-4, where the following instances of *Discriminate*$_k$ are synthesized (*k*=2…3):

```
discCompress2(HL,TL,C) ⇔
                        TL=[]
                      ∨ TL=[HTL|_] ∧ HL≠HTL
discCompress3(HL,TL,C) ⇔
                        TL=[HTL|_] ∧ HL=HTL
```

LA$_7$*(compress)* is thus as follows:

```
compress(L,C) ⇔
    L=[]       ∧ C=L                                          {E1}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ (TL=[]) ∨ (TL=[HTL|_] ∧ HL≠HTL)
               ∧ compress(TL,TC)
               ∧ C=[HL,1|TC]                    {E2,E4,E7,E8}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ TL=[HTL|_] ∧ HL=HTL
               ∧ compress(TL,TC)
               ∧ C=[HL,s(s(N))|TTC] ∧ TC=[HL,s(N)|TTC]
                                                    {E3,E5,E6}
```

Basically, the added discriminants express that the first sub-case identified at Step 6 is applicable iff *TL* does not start with an element equal to *HL*, and that the second sub-case is applicable iff *TL* does start with an element equal to *HL*.

The expanded version of *LA*$_7$*(compress)*, that is *exp(LA*$_7$*(compress))*, is as follows:

```
compress(L,C) ⇔
    L=[]       ∧ C=L
               ∧     L=[] ∧ C=[]                          {E1}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ (TL=[]) ∨ (TL=[HTL|_] ∧ HL≠HTL)
               ∧ compress(TL,TC)
               ∧ C=[HL,1|TC]
               ∧     L=[a] ∧ C=[a,1]
                   ∧ HL=a ∧ TL=[] ∧ TC=[]                 {E2}
                 ∨   L=[c,d] ∧ C=[c,1,d,1]
                   ∧ HL=c ∧ TL=[d] ∧ TC=[d,1]             {E4}
                 ∨   L=[h,i,i] ∧ C=[h,1,i,2]
                   ∧ HL=h ∧ TL=[i,i] ∧ TC=[i,2]           {E7}
                 ∨   L=[j,k,m] ∧ C=[j,1,k,1,m,1]
                   ∧ HL=j ∧ TL=[k,m] ∧ TC=[k,1,m,1]       {E8}

  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ TL=[HTL|_] ∧ HL=HTL
               ∧ compress(TL,TC)
               ∧ C=[HL,s(s(N))|TTC] ∧ TC=[HL,s(N)|TTC]
               ∧     L=[b,b] ∧ C=[b,2]
                   ∧ HL=b ∧ TL=[b] ∧ TC=[b,1]             {E3}
                 ∨   L=[e,e,e] ∧ C=[e,3]
                   ∧ HL=e ∧ TL=[e,e] ∧ TC=[e,2]           {E5}
                 ∨   L=[f,f,g] ∧ C=[f,2,g,1]
                   ∧ HL=f ∧ TL=[f,g] ∧ TC=[f,1,g,1]       {E6}
```

Note that the internal variables *HTL*, *N*, and *TTC* do not appear in the expansions.

The sample synthesis ends here. It can be shown that $LA_7(compress)$ is totally correct wrt its intended relation. ♦

Note that Steps 2,3, 5, and 6 are non-deterministic: different logic algorithms can be synthesized upon reconsideration of decisions taken at these steps.

Regarding correctness issues, Steps 2 to 7 are proven to fit the hypotheses of Theorem 7-7. It can be shown that:

$$LA_i(r) \quad \{\ll, \preceq\} \quad LA_{i-1}(r)$$
$$exp(LA_i(r)) \quad \{\ll, \not\preceq, \cong\} \quad exp(LA_{i-1}(r))$$

where $i \in \{2,\ldots,7\}$.

A careful study of this sample synthesis shows that the synthesis mechanism can actually be decomposed into two phases, an *expansion phase* (covering Steps 1 to 4) and a *reduction phase* (covering Steps 5 to 7). During the former, the expansions are gradually expanding as more variables are introduced. During the latter, the expansions are reduced by partitioning of cases.

12 The Expansion Phase

Assuming a binary intended relation $\mathcal{R}$, and its corresponding predicate $r/2$, we suppose there is a specification *EP(r)*, whose example set is defined as follows:

$$\mathcal{E}(r) = \{E_1, E_2, \ldots, E_m\} = \{\ r(x_1,y_1),\ r(x_2,y_2),\ \ldots,\ r(x_m,y_m)\ \}$$

and whose property set is defined as follows:

$$\mathcal{P}(r) = \{P_1, P_2, \ldots, P_p\} = \{\ r(s_1,t_1) \Leftarrow B_1,\ r(s_2,t_2) \Leftarrow B_2,\ \ldots,\ r(s_p,t_p) \Leftarrow B_p\ \}$$

where the body B_i of a property P_i is a conjunction of atoms. As usual in compilation, transformation, or synthesis, we assume that *EP(r)* is consistent with $\mathcal{R}$; we call this the *consistency assumption*.

Example 12-1: The *firstPlateau*/3 relation is used all along this (and the following) chapter for illustration purposes. We assume that *EP(firstPlateau)* is as follows:

```
firstPlateau([a],[a],[])                          (E1)
firstPlateau([b,b],[b,b],[])                      (E2)
firstPlateau([c,d],[c],[d])                       (E3)
firstPlateau([e,f,g],[e],[f,g])                   (E4)
firstPlateau([h,i,i],[h],[i,i])                   (E5)
firstPlateau([j,j,k],[j,j],[k])                   (E6)
firstPlateau([m,m,m],[m,m,m],[])                  (E7)
firstPlateau([X] , [X] , [] )                     (P1)
firstPlateau([X,Y],[X,Y],[] ) ⇐ X=Y               (P2)
firstPlateau([X,Y],[X] , [Y]) ⇐ X≠Y               (P3)
```

In order to keep this chapter self-sufficient, we here repeat the chosen synthesis strategy (see Section 7.3.2), where synthesis is seen as a sequence of f steps:

At Step 1, "create" $LA_1(r)$ such that:
- $LA_1(r)$ is complete wrt $\mathcal{R}$;
- $exp(LA_1(r))$ is partially correct wrt $\mathcal{R}$.

At Step i ($2 \le i \le f$), transform $LA_{i-1}(r)$ into $LA_i(r)$ such that:
- $LA_i(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_{i-1}(r)$;
- $exp(LA_i(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_{i-1}(r))$.

The following theorem (originally Theorem 7-7) indicates a practical way of pursuing this strategy at Steps 2 to f:

Let $LA(r)$ *be* $r(\boldsymbol{X}) \Leftrightarrow \vee_{1 \le j \le m} A_j$ *and* $LA'(r)$ *be* $r(\boldsymbol{X}) \Leftrightarrow \vee_{1 \le j \le m} A_j \wedge B_j$, *where* A_j, B_j *are any formulas literals. The following two assertions hold:*

(1) *If* $LA(r)$ *is complete wrt* $\mathcal{R}$ *and* $\mathcal{R}(\boldsymbol{X}) \wedge A_j \Rightarrow B_j \quad (1 \le j \le m)$
then $LA'(r)$ *is a better complete approximation of* $\mathcal{R}$ *than* $LA(r)$.

(2) *If* $LA(r)$ *is partially correct wrt* $\mathcal{R}$ *and* $A_j \Rightarrow B_j \quad (1 \le j \le m)$
then $LA'(r)$ *is a better partially correct approximation of* $\mathcal{R}$ *than* $LA(r)$.

Indeed, assertion (1) may be applied to the logic algorithms $LA_i(r)$, and assertion (2) may be applied to the logic algorithms $exp(LA_i(r))$, where $2 \le i \le f$.

Finally, we here also repeat Schema 8-3, that is version 3 of the divide-and-conquer schema:

```
R(X,Y) ⇔
          Minimal(X)       ∧ Solve(X,Y)
  ∨ ∨1≤k≤c NonMinimal(X)    ∧ Decompose(X,HX,TX)
                            ∧ Discriminate_k(HX,TX,Y)
                            ∧ (     SolveNonMin_k(HX,TX,Y)
                                |
                                    R(TX,TY)
                                ∧ Process_k(HX,HY)
                                ∧ Compose_k(HY,TY,Y)       )
```

Instantiating some predicate variable(s) of this schema is a synthesis step. The synthesis mechanism is expressed as the following sequence of f=7 steps (see Example 8-10):

- Step 1: Syntactic creation of a first approximation;
- Step 2: Synthesis of *Minimal* and *NonMinimal*;
- Step 3: Synthesis of *Decompose*;
- Step 4: Syntactic introduction of the recursive atoms;
- Step 5: Synthesis of *Solve* and the $SolveNonMin_k$;
- Step 6: Synthesis of the $Process_k$ and $Compose_k$;
- Step 7: Synthesis of the $Discriminate_k$.

The *expansion phase* of synthesis comprises the first four steps of this mechanism. In Section 12.1 to Section 12.4, we give complete descriptions of each of these steps: we state their objectives, describe the used methods, analyze their correctness and progression behaviors, and illustrate them on the *firstPlateau*/3 and *compress*/2 relations.

12.1 Step 1: Syntactic Creation of a First Approximation

A first approximation needs to be "created" so as to establish the applicability of Theorem 7-7.

12.1.1 Objective

According to the chosen strategy, the objective at Step 1 is to "create" $LA_1(r)$ such that:

(1) $LA_1(r)$ is complete wrt $\mathcal{R}$;
(2) $exp(LA_1(r))$ is partially correct wrt $\mathcal{R}$.

The achievement of this objective makes Theorem 7-7 applicable for the subsequent steps.

12.1.2 Method

The safest method to achieve this is to postulate that $LA_1(r)$ is $\top_r$, which is as follows:

```
r(X,Y) ⇔
    true                                              {E1,E2,…,Em}
```

This is always possible, hence Step 1 is fully deterministic (and thus never fails). Moreover, $LA_1(r)$ is also fully deterministic. Step 1 yields a unique case, called the *all-true case*. As usual, the set annotation to a disjunct explains which examples are covered by that disjunct. Now, $exp(LA_1(r))$ is as follows:

```
r(X,Y) ⇔
    true      ∧     X=x1 ∧ Y=y1                          {E1}
                ∨   X=x2 ∧ Y=y2                          {E2}
                ∨   ...                                   ...
                ∨   X=xm ∧ Y=ym                          {Em}
```

```
firstPlateau(L,P,S) ⇔
    true                                                {E1,…,E7}
```

Logic Algorithm 12-1: *LA_1(firstPlateau)*

```
firstPlateau(L,P,S) ⇔
    true
              ∧     L=[a] ∧ P=[a] ∧ S=[]                {E1}
                ∨   L=[b,b] ∧ P=[b,b] ∧ S=[]            {E2}
                ∨   L=[c,d] ∧ P=[c] ∧ S=[d]             {E3}
                ∨   L=[e,f,g] ∧ P=[e] ∧ S=[f,g]         {E4}
                ∨   L=[h,i,i] ∧ P=[h] ∧ S=[i,i]         {E5}
                ∨   L=[j,j,k] ∧ P=[j,j] ∧ S=[k]         {E6}
                ∨   L=[m,m,m] ∧ P=[m,m,m] ∧ S=[]        {E7}
```

Logic Algorithm 12-2: *exp(LA_1(firstPlateau))*

12.1.3 Correctness

We can now prove the following theorem establishing that Step 1 is in line with the strategy:

Theorem 12-1: *Step* 1 *achieves the two conditions of its objective:*

(1) *$LA_1(r)$ is complete wrt $\mathcal{R}$;*

(2) *exp($LA_1(r)$) is partially correct wrt $\mathcal{R}$.*

Proof 12-1: Let's prove these assertions one by one:

(1) Trivial (by construction).

(2) We obviously have that *exp($LA_1(r)$)* is totally correct wrt $\mathcal{E}(r)$. By the consistency assumption, *exp($LA_1(r)$)* is thus partially correct wrt $\mathcal{R}$. ❑

Note that this proof is independent of $\mathcal{R}$.

12.1.4 Illustration

Let's illustrate Step 1 on the *firstPlateau*/3 and *compress*/2 relations.

Example 12-2: For the *firstPlateau*/3 relation, Step 1 yields *LA_1(firstPlateau)* as shown in Logic Algorithm 12-1, and *exp(LA_1(firstPlateau))* as shown in Logic Algorithm 12-2.

Example 12-3: For the *compress*/2 relation, Step 1 yields a logic algorithm and its expansion as shown in Example 11-1.

12.2 Step 2: Synthesis of *Minimal* and *NonMinimal*

An instantiation of the *Minimal* (respectively *NonMinimal*) predicate variable tests whether the induction parameter is of a minimal (respectively non-minimal) form.

12.2.1 Objective

The objective at Step 2 is to instantiate the predicate variables *Minimal* and *NonMinimal* of the divide-and-conquer schema. This amounts to transforming *$LA_1(r)$* into *$LA_2(r)$* such that it is covered by the following schema:

```
r(X,Y) ⇔
    Minimal(X)
  ∨ NonMinimal(X)
```

where X is the selected induction parameter. This amounts to splitting the all-true case into two cases, called the *minimal case* and the *non-minimal case*, respectively. Let *minimal*/1 and *nonMinimal*/1 be the synthesized instances. The following objectives of the strategy:

(1) $LA_2(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_1(r)$;

(2) $exp(LA_2(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_1(r))$;

and the following integrity constraints of the divide-and-conquer schema:

(3) X is single and of an inductive type;

(4) $X \in dom(r) \Rightarrow minimal(X) \vee nonMinimal(X)$;

must be satisfied.

12.2.2 Method

This objective can be achieved by the following sequence of tasks:

- Task A: selection of an induction parameter;
- Task B: ordering of the examples into a sequence;
- Task C: partitioning of the example sequence into 2 sub-sequences;
- Task D: selection of *minimal*/1 and *nonMinimal*/1.

Let's explain these tasks one by one.

Task A: Selection of an Induction Parameter

Task A is done in the same fashion as in manual logic algorithm construction [Deville 90] (see Section 4.2.1): the induction parameter must be simple, and of an inductive type. This selection can be automated by type inference from the given examples. In case more detailed specification knowledge is available, the Functionality Heuristic (Heuristic 4-1) and the Directionality Heuristic (Heuristic 4-2) may even be used, the latter being of higher precedence in case they yield contradictory results. A reasonable implementation of this synthesis mechanism would actually even accept preference hints from the specifier. We assume that the u^{th} parameter is selected as induction parameter, where $1 \leq u \leq n$.

Example 12-4: Within the *firstPlateau*/3 relation, all three parameters are of an inductive type, namely *list*. If we consider the specification of Figure 4-2, both heuristics would point towards L being the "best" induction parameter. Without this extended specification knowledge, a non-deterministic selection of an induction parameter among $\{L, P, S\}$ takes place. We assume that L is effectively selected: its position u is 1.

Task B: Ordering of the Examples

Task B is based on the notion of term size. As agreed in Section 11.2, this theoretical discussion only considers the type-constructing functors s/1 (for integers) and •/2 (for lists).

Definition 12-1: The *size* of a ground term t, denoted $\#t$, is the number of appearances of the outmost type-constructing functor within t, if t is of an inductive type, and *undefined* otherwise.

Example 12-5: For instance, $\#3 = \#s(s(s(0))) = 3$, because s/1 appears thrice in 3. Also, $\#[a,b] = \#\bullet(a, \bullet(b, nil)) = 2$, because •/2 appears twice in $[a,b]$. But $\#a = undefined$, because no type-constructing functor occurs in a.

Definition 12-2: Given an example E of predicate r/n, and an integer position i (where $1 \leq i \leq n$), the size of E wrt i, denoted $\#(E,i)$, is the size of the i^{th} parameter of E.

Example 12-6: Considering the examples of *EP(firstPlateau)* (see Example 12-1), we have $\#(E_1,1) = 1$, and $\#(E_2,1) = \#(E_3,1) = 2$, and $\#(E_4,1) = \#(E_5,1) = \#(E_6,1) = \#(E_7,1) = 3$.

The examples of $\mathcal{E}(r)$ are now ordered by increasing-or-equal size of the examples wrt the position u of the selected induction parameter. The result is a sequence $\mathcal{S}$ of examples.

Example 12-7: For the *firstPlateau*/3 relation, with induction on L, the ordered sequence is exactly the one of the given example set: $\mathcal{S} = [E_1,E_2,E_3,E_4,E_5,E_6,E_7]$.

Task C: Partitioning of the Examples

Task C non-deterministically partitions the sequence $\mathcal{S}$ into 2 sub-sequences $\mathcal{S}_1$ and $\mathcal{S}_2$, such that:

$$\forall E_1 \in \mathcal{S}_1 \;\; \forall E_2 \in \mathcal{S}_2 \quad \#(E_1,u) < \#(E_2,u)$$

The idea is that the examples of $\mathcal{S}_1$ are hypothesized to be of the minimal form, while the examples of $\mathcal{S}_2$ are hypothesized to be of the non-minimal form. Hence the partitioning into two sub-sequences, since we agreed upon a total of two forms. We call the elements of $\mathcal{S}_1$ the *minimal examples*, and the elements of $\mathcal{S}_2$ the *non-minimal examples*.

Example 12-8: For the *firstPlateau*/3 relation, with induction on L, there are two ways of partitioning the example sequence $\mathcal{S}$: either $\mathcal{S}_1 = [E_1]$ and $\mathcal{S}_2 = [E_2,E_3,E_4,E_5,E_6,E_7]$, or $\mathcal{S}_1 = [E_1,E_2,E_3]$ and $\mathcal{S}_2 = [E_4,E_5,E_6,E_7]$. We assume that the first partition is selected.

Task D: Selection of minimal/1 *and nonMinimal*/1

Task D consists of mechanically extracting *minimal*/1 and *nonMinimal*/1 from a parameterized database. Suppose that the sizes of the examples in $\mathcal{S}_1$ are within the integer interval $[p_1,\ldots,q_1]$, and that the sizes of the examples in $\mathcal{S}_2$ are within the integer interval $[p_2,\ldots,q_2]$. Table 12-1 below shows the used fragment of the database, which only uses primitive predicates (for simplicity, we here restrict it to the situation where $p = p_1 = q_1$ and $p_2 = q_1 + 1$):

Table 12-1: Excerpt of the database of instances of *Minimal*/1 and *NonMinimal*/1

Type of X	*Minimal(X)*	*NonMinimal(X)*
integer	$X = p$	$X > p$
list	$X = [\_,\ldots,\_]$ (p elements)	$X = [\_,\ldots,\_,\_\|\_]$ ($> p$ elements)

Type inference from the examples may be used to access the appropriate row of the database. The hypothesis underlying this excerpt of the database is that the examples of sub-sequence $\mathcal{S}_1$ represent all the sizes of the minimal form, whereas the examples of sub-sequence $\mathcal{S}_2$ represent only a prefix of the sizes of the non-minimal form. This amounts to assuming that q_2 may be safely replaced by $+\infty$. This hypothesis is quite reasonable and covers a huge number of circumstances. Instances reflecting other hypotheses may also be stored in the database. We call such hypotheses the *domain extrapolation hypotheses*. The mechanical instance extraction is thus non-deterministic. The commonality between all instances is that there is only concern about the size of the induction parameter, and not about its value (hence the presence of anonymous variables in the database). The mentioned hypotheses have an obvious influence upon the formulation of a methodology for choosing "good" examples.

Note that $exp(LA_2(r))$ is as follows:

$$\begin{array}{l} \texttt{r(X,Y)} \Leftrightarrow \\ \qquad \texttt{minimal(X)} \;\wedge\; \vee_{j\in I} \; \texttt{X=x}_j \wedge \texttt{Y=y}_j \\ \quad \vee \;\; \texttt{nonMinimal(X)} \;\wedge\; \vee_{j\in J} \; \texttt{X=x}_j \wedge \texttt{Y=y}_j \end{array}$$

where I is the set of indices of the minimal examples, and J is the set of indices of the non-minimal examples, such that $I \cup J$ is a permutation of the integer set $\{1,\ldots,m\}$.

Example 12-9: For the *firstPlateau*/3 relation, with the decisions taken by the previous tasks, we obtain $p = p_1 = q_1 = 1$, and $p_2 = 2 = q_1 + 1$. Following the hypothesis underlying Table 12-1, we assume that the minimal form is [_] (that is, a list of exactly one element) and that the non-minimal form is [_,_|_] (that is, a list of at least two elements). The instances of *Minimal* and *NonMinimal* thus are L = [_], and L = [_,_|_], respectively.

Comments

This ends the presentation of the method used at Step 2. Before the discussion of its correctness, a few comments are necessary.

First, note that L = [_] is strictly speaking not an instance of *Minimal(X)*. The predicate variable *Minimal*/1 is rather instantiated to, say, a predicate *minimal*/1, which is defined by the following logic algorithm: *minimal(X)* $\Leftrightarrow$ X = [_]. It is actually only the renaming substitution {*X*/*L*} and then the unfolding of *minimal(L)* that bring the atom L = [_] into some logic algorithm for *firstPlateau*/3. So it is only by abuse of language that we may speak of L = [_] as being an "instance" of *Minimal(X)*.

Second, it is the very focus on structural aspects of the induction parameter (its size, that is) that makes this database approach possible. If semantic aspects of the induction parameter (its value, that is) also have to be taken into account, then a deductive approach reasoning backwards from instances of all other predicate variables becomes necessary [Smith 85]. We here clearly separate these aspects: the structure of the induction parameter is analyzed at Step 2 for instantiating *Minimal* and *NonMinimal*, and the value of the induction parameter is analyzed at Step 7 for instantiating the $Discriminate_k$.

Finally, the methods of tasks A, C, and D are non-deterministic, but finite. This means that choice-points are created there, and that the made selections may be reconsidered later (either because synthesis fails, or because synthesis succeeds and the specifier wants more algorithms). Only Task A could possibly fail, namely if there is no parameter of an inductive type. Do not confuse however non-deterministic synthesis and a non-deterministic synthesized algorithm. The latter would feature either mutually non-exclusive cases or predicates that are non-deterministic in the all-ground mode. The logic algorithm synthesized at Step 2 is deterministic, because the two cases are mutually exclusive by construction, and because the introduced predicates are deterministic in the all-ground mode.

12.2.3 Correctness

We can now prove the following theorem establishing that Step 2 is in line with the strategy:

Theorem 12-2: *Step 2 achieves the four conditions of its objective:*

(1) $LA_2(r)$ *is a better complete approximation of* $\mathcal{R}$ *than* $LA_1(r)$*;*
(2) $exp(LA_2(r))$ *is a better partially correct approximation of* $\mathcal{R}$ *than* $exp(LA_1(r))$*;*
(3) *X is single and of an inductive type;*
(4) $X \in dom(\mathcal{R}) \Rightarrow minimal(X) \vee nonMinimal(X)$.

Proof 12-2: Let's prove these assertions one by one:

(1) By assertion (1) of Theorem 7-7 and assertion (1) of Theorem 12-1, it suffices to show that:

$$\mathcal{R}(X,Y) \wedge true \Rightarrow minimal(X) \vee nonMinimal(X)$$

in order to prove that $LA_2(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_1(r)$. This statement is actually implied by Point (4) of this theorem, because $\mathcal{R}(X,Y) \Rightarrow X \in dom(\mathcal{R})$ and $F \vee G \Rightarrow F \vee G$.

(2) By assertion (2) of Theorem 7-7 and assertion (2) of Theorem 12-1, it suffices to show that:

```
firstPlateau(L,P,S) ⇔
    L=[_]
  ∨ L=[_,_|_]
```

Logic Algorithm 12-3: *LA_2(firstPlateau)*

```
firstPlateau(L,P,S) ⇔
    L=[_]
                ∧     L=[a] ∧ P=[a] ∧ S=[]          {E1}
  ∨ L=[_,_|_]
                ∧     L=[b,b] ∧ P=[b,b] ∧ S=[]      {E2}
                  ∨   L=[c,d] ∧ P=[c] ∧ S=[d]       {E3}
                  ∨   L=[e,f,g] ∧ P=[e] ∧ S=[f,g]   {E4}
                  ∨   L=[h,i,i] ∧ P=[h] ∧ S=[i,i]   {E5}
                  ∨   L=[j,j,k] ∧ P=[j,j] ∧ S=[k]   {E6}
                  ∨   L=[m,m,m] ∧ P=[m,m,m] ∧ S=[]  {E7}
```

Logic Algorithm 12-4: *exp(LA_2(firstPlateau))*

$$X{=}x_j \wedge Y{=}y_j \Rightarrow minimal(X) \quad (j \in I)$$
$$X{=}x_j \wedge Y{=}y_j \Rightarrow nonMinimal(X) \quad (j \in J)$$

in order to prove that *exp(LA_2(r))* is a better partially correct approximation of $\mathcal{R}$ than *exp(LA_1(r))*. This is achieved by the methods of Tasks B to D.

(3) By the method of Task A, the selected induction parameter is single and of an inductive type.

(4) By the methods of Tasks B to D, *minimal*/1 and *nonMinimal*/1 are mutually exclusive over the domain of the induction parameter within $\mathcal{E}(r)$. If the right domain extrapolation hypothesis is taken by the method of Task D, then *minimal*/1 and *nonMinimal*/1 are also mutually exclusive over the entire intended domain of the induction parameter. ❑

This proof hinges on the fact that the right domain extrapolation hypothesis must be taken by the method of Task D. In the sequel, we assume that such is the case. Of course, a reasonable implementation of this synthesis mechanism would include some interaction with the specifier so as to maximize the confidence that everything goes right.

12.2.4 Illustration

Let's illustrate Step 2 on the *firstPlateau*/3 and *compress*/2 relations.

Example 12-10: For the *firstPlateau*/3 relation, Step 2 proceeds as illustrated in the previous six examples. Logic Algorithm 12-3 shows *LA_2(firstPlateau)*, whereas Logic Algorithm 12-4 shows *exp(LA_2(firstPlateau))*.

Example 12-11: For the *compress*/2 relation, Step 2 proceeds as follows. At Task A, both parameters are found to be of an inductive type, namely *list*. If we considered the specification of Figure 4-1, the heuristics would point towards *L* being the "best" induction parameter. Without this extended specification knowledge, a non-deterministic selection of an induction parameter among {*L, C*} takes place. We assume that *L* is effectively selected: its position *u* is 1. At Task B, we have $\#(E_1,1) = 0$, and $\#(E_2,1) = 1$, and $\#(E_3,1) = \#(E_4,1) = 2$, and $\#(E_5,1) = \#(E_6,1) = \#(E_7,1) = \#(E_8,1) = 3$. So the ordered sequence of examples is exactly the one of the given example set: $\mathcal{S} = [E_1,E_2,E_3,E_4,E_5,E_6,E_7,E_8]$. At Task C, there are three ways of partitioning this sequence $\mathcal{S}$: either $\mathcal{S}_1 = [E_1]$ and $\mathcal{S}_2 = [E_2,E_3,E_4,E_5,E_6,E_7,E_8]$,

or $\mathcal{S}_1 = [E_1,E_2]$ and $\mathcal{S}_2 = [E_3,E_4,E_5,E_6,E_7,E_8]$, or $\mathcal{S}_1 = [E_1,E_2,E_3,E_4]$ and $\mathcal{S}_2 = [E_5,E_6,E_7,E_8]$. We assume that the first partition is selected. At Task D, we obtain $p = p_1 = q_1 = 0$, as well as $p_2 = 1 = q_1 + 1$. Following the hypothesis underlying Table 12-1, we assume that the minimal form is [] (that is, an empty list) and that the non-minimal form is [_|_] (that is, a list of at least one element). The corresponding instances of *Minimal* and *NonMinimal* thus are $L = []$, and $L = [\_|\_]$, respectively. The resulting logic algorithm and its expansion are shown in Example 11-1.

12.3 Step 3: Synthesis of *Decompose*

An instantiation of the *Decompose* predicate variable deterministically decomposes, in the non-minimal case, the induction parameter X into a vector $\boldsymbol{HX}$ of heads and a vector $\boldsymbol{TX}$ of tails, each tail TX_i being smaller than X according to some well-founded relation. These tails are meant for the recursive computation of the tails TY_i of the other parameter Y.

12.3.1 Objective

The objective at Step 3 is to instantiate the predicate variable *Decompose* of the divide-and-conquer schema. This amounts to transforming $LA_2(r)$ into $LA_3(r)$ such that it is covered by the following schema:

```
r(X,Y) ⇔
    minimal(X)
  ∨ nonMinimal(X) ∧ Decompose(X,HX,TX)
```

Let *decompose* be the synthesized instance. It is of arity $1+h+t$, where $h = \#\boldsymbol{HX}$ and $t = \#\boldsymbol{TX}$. The following objectives of the strategy:

(1) $LA_3(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_2(r)$;

(2) $exp(LA_3(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_2(r))$;

and the following integrity constraints of the divide-and-conquer schema:

(3) $h \geq 0$ and $t \geq 1$;

(4) $X \in dom(\mathcal{R}) \wedge nonMinimal(X) \Rightarrow \exists!\boldsymbol{HX}\, \exists!\boldsymbol{TX}\ decompose(X,\boldsymbol{HX},\boldsymbol{TX})$;

(5) $\exists$"<" $\forall(decompose(X,\boldsymbol{HX},TX_1,\ldots,TX_t) \Rightarrow \forall i \in \{1,\ldots,t\}\ TX_i$ "<" $X)$;

must be satisfied.

12.3.2 Method

This objective can be achieved by the following sequence of tasks:

- Task E: selection of a decomposition strategy;
- Task F: selection of *decompose*/$1+h+t$.

Let's explain these tasks one by one.

Task E: Selection of a Decomposition Strategy

Task E consists of selecting one of the three decomposition strategies seen in Section 5.2.3. As a reminder, these strategies are:

- *intrinsic decomposition*: X is decomposed into $h \geq 1$ heads and $t \geq 1$ tails in a manner reflecting the definition of the type of X;
- *extrinsic decomposition*: X is decomposed into $h \geq 0$ heads and $t \geq 1$ tails in a manner reflecting the definition of the type of some other parameter than X, or reflecting the intended relation;
- *logarithmic decomposition*: X is decomposed into $h = 0$ heads and $t \geq 2$ tails of about equal size.

An intrinsic decomposition reflects a well-founded relation selected via the Intrinsic Heuristic, and an extrinsic or logarithmic decomposition reflects a well-founded relation selected via the Extrinsic Heuristic (see Chapter 4). The selection of a strategy is a high-level design decision that may significantly affect the complexity of the resulting algorithm (but probably not its existence). A reasonable implementation of this synthesis mechanism would accept a preference hint from the specifier.

If an intrinsic decomposition is selected, then a *decomposition decrement*, denoted d, needs to be selected before proceeding to Task F. This is done by selecting a value within the following integer interval:

$$[\ \#(E_w,u) - \#(E_v,u),\ldots,\#(E_w,u)\ ]$$

where E_v is the last (largest-sized) example of $\mathcal{S}_1$, and E_w is the first (smallest-sized) example of $\mathcal{S}_2$. The lower bound represents the size decrease needed to decompose an example of the size of E_v into an example of the size of E_w. The upper bound is justified by the fact that one can at most take away as many elements as there are elements in the smallest non-minimal example. Note that d is thus constrained to be a constant.

Example 12-12: For the *firstPlateau*/3 relation, suppose the intrinsic decomposition strategy is selected, and that the decomposition decrement d is selected to be 1, namely from the interval [1,…,2].

Task F: Selection of decompose/1+h+t

Task F consists of mechanically extracting *decompose*/1+h+t from a parameterized database. Tables 12-2 to 12-4 below show the used fragments of the database, which only uses primitive predicates (we omit the subscripts i from the HX_i and TX_i in case $h = 1$ or $t = 1$):

Table 12-2: Excerpt of the database of intrinsic instances of *Decompose*/1+h+t

Type of X	*Decompose(X,**HX**,**TX**)*
integer	$add(TX,d,X) \wedge HX{=}X$
list	$X = [HX_1,\ldots,HX_d \mid TX]$

Table 12-3: Excerpt of the database of extrinsic instances of *Decompose*/1+h+t

Type of X	*Decompose(X,**HX**,**TX**)*
integer	—
list	*firstPlateau(X,HX,TX)* …
integer-list	$X = [HX \mid T] \wedge partition(T,HX,TX_1,TX_2)$ …

Table 12-4: Excerpt of the database of logarithmic instances of *Decompose*/1+h+t

type of X	*Decompose(X,**HX**,**TX**)*
integer	—
list	$split(X,TX_1,TX_2)$ …

Type inference from the examples may be used to access the appropriate row of the database. Note that *exp(LA$_3$(r))* is as follows:

```
r(X,Y) ⇔
        minimal(X)    ∧     ∨j∈I     X=xj ∧ Y=yj
     ∨  nonMinimal(X) ∧ decompose(X,HX,TX)
                      ∧     ∨j∈J     X=xj ∧ Y=yj
                                   ∧ HX=hxj ∧ TX=txj
```

where the $\boldsymbol{hx}_j$ and $\boldsymbol{tx}_j$ are vectors of witnesses such that *decompose*(x_j,$\boldsymbol{hx}_j$,$\boldsymbol{tx}_j$) holds ($j \in \mathcal{J}$).

Example 12-13: For the *firstPlateau*/3 relation, with the decisions taken at the previous task, the instance $L = [HL|TL]$ is extracted from Table 2. This means that $h = t = 1$.

Comments

This ends the presentation of the method used at Step 3. Before the discussion of its correctness, a few comments are necessary.

First, the support of the extrinsic and logarithmic decomposition strategies is quite difficult, as the required knowledge about the intended relation is not available. This apparent drawback may be alleviated however by the observation that the logic algorithms *LA(r-int-X)* and *LA(r-ext-Y)* are by construction quite similar. Compare *LA(compress)* (LA 4-1) and *LA(compress-ext-C)* (LA 5-3), or *LA(compress-int-C)* (LA 5-2) and *LA(compress-ext-L)* (LA 5-1) for illustrations of this observation. In other words, the support of the extrinsic and logarithmic decomposition strategies often doesn't give rise to new logic algorithms. This is due to the reversibility of logic algorithms: *Compose(**HY**,**TY**,Y)* may not only be used in the *composition mode* (***ground***,***ground***,*var*), but also in the *decomposition mode* (***var***,***var***, *ground*). Similarly, *Decompose(X,**HX**,**TX**)* may not only be used in the *decomposition mode* (*ground*,***var***,***var***), but also in the *composition mode* (*var*,***ground***,***ground***). However, the constraints (4) and (5) sometimes prevent the reversion of *LA(r-int-X)* into *LA(r-ext-Y)*. Indeed, the decomposition mode of *Compose(**HY**,**TY**,Y)* might be non-deterministic. For instance, *efface(HY,Y,TY)*, *insert(HY,TY,Y)*, *append*(TY_1,[$HY|TY_2$],*Y*), and *merge*(TY_1,TY_2,*Y*) may not be used as instances of *Decompose*, but are suitable as instances of *Compose*. Or the decomposition mode of *Compose(**HY**,**TY**,Y)* might not reflect a well-founded relation. For instance, $Y = TY_i$ may not be used as an instance of *Decompose*, but is suitable as an instance of *Compose*. Such an instance may appear because *Y* is an auxiliary parameter, or because *Y* would be part of a compound induction parameter for the reversion.

Second, note that the methods of both tasks are non-deterministic, finite, and can never fail. However, the introduced predicates are deterministic in the decomposition mode. This implies that the witnesses $\boldsymbol{hx}_j$ and $\boldsymbol{tx}_j$ are unique. The logic algorithm synthesized at Step 3 is thus deterministic.

Third, the methods used by Steps 2 and 3 could be abstracted into a *Database Method*. Roughly speaking, this would work as follows: given an example set $\mathcal{E}$, some computations *C*, and a database *DB*, select an entry from *DB* using the key generated by executing *C* on $\mathcal{E}$.

Finally, with an intrinsic decomposition, obvious simplification opportunities arise between *nonMinimal* and *decompose*. However, we never perform these simplifications, so as to present canonical representations of all logic algorithms.

12.3.3 Correctness

We can now prove the following theorem establishing that Step 3 is in line with the strategy:

Theorem 12-3: *Step* 3 *achieves the five conditions of its objective:*

(1) *LA*$_3$*(r) is a better complete approximation of* $\mathcal{R}$ *than LA*$_2$*(r);*

(2) *exp(LA$_3$(r)) is a better partially correct approximation of $\mathcal{R}$ than exp(LA$_2$(r));*
(3) $h \geq 0$ *and* $t \geq 1$*;*
(4) $X \in dom(\mathcal{R}) \wedge nonMinimal(X) \Rightarrow \exists!\boldsymbol{HX}\, \exists!\boldsymbol{TX}$ *decompose(X,**HX**,**TX**);*
(5) $\exists$"<" $\forall$*(decompose(X,**HX**,TX$_1$,…,TX$_t$)* $\Rightarrow$ $\forall i \in \{1,\ldots,t\}$ *TX$_i$* "<" *X).*

Proof 12-3: Let's prove these assertions one by one:

(1) By assertion (1) of Theorem 7-7 and assertion (1) of Theorem 12-2, it suffices to show that:

$$\mathcal{R}(X,Y) \wedge minimal(X) \Rightarrow true$$

$$\mathcal{R}(X,Y) \wedge nonMinimal(X) \Rightarrow decompose(X,\boldsymbol{HX},\boldsymbol{TX})$$

in order to prove that *LA$_3$(r)* is a better complete approximation of $\mathcal{R}$ than *LA$_2$(r)*. The first statement is trivially true (and we shall omit such statements in the sequel). The second statement is implied by Point (4) of this theorem.

(2) By assertion (2) of Theorem 7-7 and assertion (2) of Theorem 12-2, it suffices to show that:

$$minimal(X) \wedge X{=}x_j \wedge Y{=}y_j \Rightarrow true \quad (j \in I)$$

$$nonMinimal(X) \wedge X{=}x_j \wedge Y{=}y_j \Rightarrow$$
$$decompose(X,\boldsymbol{HX},\boldsymbol{TX}) \wedge \boldsymbol{HX}{=}\boldsymbol{hx}_j \wedge \boldsymbol{TX}{=}\boldsymbol{tx}_j \quad (j \in \mathcal{J})$$

in order to prove that *exp(LA$_3$(r))* is a better partially correct approximation of $\mathcal{R}$ than *exp(LA$_2$(r))*. The first statement is trivially true (and we shall omit such statements in the sequel). The second statement is implied by Point (4) of this theorem.

(3) The databases used by the method of Task F guarantee that $h \geq 0$ and $t \geq 1$.
(4) The databases used by the method of Task F guarantee that each instance is deterministic.
(5) The databases used by the method of Task F guarantee that each instance reflects decomposition according to some well-founded relation. ❑

12.3.4 Illustration

Let's illustrate Step 3 on the *firstPlateau*/3 and *compress*/2 relations.

Example 12-14: For the *firstPlateau*/3 relation, Step 3 proceeds as illustrated in the previous two examples. Logic Algorithm 12-5 shows *LA$_3$(firstPlateau)*, whereas Logic Algorithm 12-6 shows *exp(LA$_3$(firstPlateau))*.

Example 12-15: For the *compress*/2 relation, Step 3 proceeds as follows. At Task E, suppose the intrinsic decomposition strategy is selected. The decomposition decrement d is then necessarily 1, because selected from the singleton interval [1,…,1]. At Task F, the instance $L = [HL|TL]$ is extracted from Table 2. This means that $h = t = 1$. The resulting logic algorithm and its expansion are shown in Example 11-1.

12.4 Step 4: Syntactic Introduction of the Recursive Atoms

In the non-minimal case, some tails ***TY*** of *Y* are obtained by recursion on the tails ***TX*** of *X*.

12.4.1 Objective

The objective at Step 4 is to instantiate the conjunction of recursive atoms of the divide-and-conquer schema. This amounts to transforming *LA$_3$(r)* into *LA$_4$(r)* such that it is covered by the following schema:

```
firstPlateau(L,P,S) ⇔
    L=[_]
  v L=[_,_|_] ∧ L=[HL|TL]
```

Logic Algorithm 12-5: *LA_3(firstPlateau)*

```
firstPlateau(L,P,S) ⇔
    L=[_]
                ∧     L=[a] ∧ P=[a] ∧ S=[]                {E1}
  v L=[_,_|_] ∧ L=[HL|TL]
                ∧     L=[b,b] ∧ P=[b,b] ∧ S=[]
                    ∧ HL=b ∧ TL=[b]                       {E2}
                  v   L=[c,d] ∧ P=[c] ∧ S=[d]
                    ∧ HL=c ∧ TL=[d]                       {E3}
                  v   L=[e,f,g] ∧ P=[e] ∧ S=[f,g]
                    ∧ HL=e ∧ TL=[f,g]                     {E4}
                  v   L=[h,i,i] ∧ P=[h] ∧ S=[i,i]
                    ∧ HL=h ∧ TL=[i,i]                     {E5}
                  v   L=[j,j,k] ∧ P=[j,j] ∧ S=[k]
                    ∧ HL=j ∧ TL=[j,k]                     {E6}
                  v   L=[m,m,m] ∧ P=[m,m,m] ∧ S=[]
                    ∧ HL=m ∧ TL=[m,m]                     {E7}
```

Logic Algorithm 12-6: *exp(LA_3(firstPlateau))*

```
r(X,Y) ⇔
        minimal(X)
     v  nonMinimal(X)   ∧ decompose(X,HX,TX)
                        ∧ (     true
                            v
                                r(TX,TY)              )
```

This amounts to splitting the non-minimal case into two non-mandatory cases, called the *non-recursive non-minimal case* and the *recursive non-minimal case*, respectively. For convenience, we drop the qualifier "non-minimal" from these two names. Recursion may be useless in the sense that the recursively computed ***TY*** are not needed for the computation of *Y*. Useless recursion wouldn't affect the correctness of a logic algorithm, though. Its elimination is thus rather a matter of algorithm optimization. The following objectives of the strategy:

(1) *LA_4(r)* is a better complete approximation of $\mathcal{R}$ than *LA_3(r)*;

(2) *exp(LA_4(r))* is a better partially correct approximation of $\mathcal{R}$ than *exp(LA_3(r))*;

must be satisfied.

12.4.2 Method

The case splitting and the introduction of a conjunction of recursive atoms into one of the resulting cases are mere syntactic operations, and hence pretty straightforward.

However, the decision whether both cases are necessary involves some computations on the non-minimal examples during the generation of *exp(LA_4(r))*. Recall from Definition 7-25 that *exp* is actually a ternary operator that expands the disjuncts of a logic algorithm *LA(r)* wrt some covered examples $\mathcal{E}$*(r)* and some oracle *O(r)*. We agreed on the default usage of the so-called deductive oracle based on the specification *EP(r)*. This oracle is automatable, sound (under the consistency assumption), but not necessarily complete. The deductive ora-

cle could even be based on *LA(r)* itself: it would be automated, sound (if *LA(r)* is partially correct wrt $\mathcal{R}$), and complete (if *LA(r)* is complete wrt $\mathcal{R}$). But such an oracle would be rarely useful in practice. Other automated oracles can be imagined, performing other types of reasoning on the examples and properties, such as analogical reasoning. A reasonable implementation of this synthesis mechanism would also interact with the specifier, especially for queries where all other oracles yield empty witness sets. Such a *human oracle* is supposed to be sound and complete.

Now, the decision whether both cases are really necessary is achieved by the following sequence of four tasks:

- Task G: compute $exp(LA_4(r))$ according to some sound oracle *O(r)*;
- Task H: in the non-recursive case, eliminate every disjunct that is an expansion wrt some example that is also covered by the recursive case. Indeed, every example covered by the recursive case in $LA_4(r)$ is also covered by the non-recursive case (but the opposite doesn't hold): this task amounts to eliminating unwanted redundancy;
- Task I: in the recursive case, remove every disjunct where the general example *procComp(<**hx**$_j$>,<**ty**$_j$>,y_j)* would have no admissible alternatives[18] wrt $a = b = 1$, and expand the non-recursive case wrt the corresponding example E_j. Indeed, if y_j is not somehow constructed in terms of **ty**$_j$, then recursion is useless for testing example E_j.

This amounts so far to splitting the sequence of examples $\mathcal{S}_2$ (see Step 2) into two complementary (but possibly empty) sub-sequences $\mathcal{S}_{21}$ and $\mathcal{S}_{22}$, such that the examples of $\mathcal{S}_{21}$ are covered by the disjuncts of the non-recursive case, whereas the examples of $\mathcal{S}_{22}$ are covered by the disjuncts of the recursive case. We call the elements of $\mathcal{S}_{21}$ the *non-recursive examples*, and the elements of $\mathcal{S}_{22}$ the *recursive examples*.

- Task J: take the following decisions:
  - if $\mathcal{S}_{21}$ is empty, then conjecture that recursion is necessary for every possible example of $\mathcal{R}$, rather than just for those of $\mathcal{E}(r)$, and eliminate the non-recursive case in both $LA_4(r)$ and $exp(LA_4(r))$;
  - if $\mathcal{S}_{22}$ is empty, then conjecture that recursion is useless for every possible example of $\mathcal{R}$, rather than just for those of $\mathcal{E}(r)$, and eliminate the recursive case in both $LA_4(r)$ and $exp(LA_4(r))$.

  If the recursive case is eliminated, then this synthesis mechanism is overkill.

In the sequel, we assume that the *exp*/3 operator actually executes all of these four tasks, rather than just the first one. Note that $exp(LA_4(r))$ is then as follows:

```
r(X,Y) ⇔
        minimal(X)     ∧      ∨j∈I      X=xj ∧ Y=yj
    ∨   nonMinimal(X) ∧ decompose(X,HX,TX)
                       ∧      ∨j∈K      X=xj ∧ Y=yj
                                        ∧ HX=hxj ∧ TX=txj
    ∨   nonMinimal(X) ∧ decompose(X,HX,TX)
                       ∧ r(TX,TY)
                       ∧      ∨j∈L      X=xj ∧ Y=yj
                                        ∧ HX=hxj ∧ TX=txj
                                        ∧ TY∈tyj
```

where $\mathcal{K}$ is the set of indices of the non-recursive examples, and $\mathcal{L}$ is the set of indices of the recursive examples, such that $\mathcal{K} \cup \mathcal{L}$ is a permutation of $\mathcal{J}$. The **ty**$_j$ are vectors of sets ty_{jk} of witnesses such that $r(tx_{jk},ty_{jki})$ holds for some alternative ty_{jki} of ty_{jk} ($j \in \mathcal{L}$, $1 \leq k \leq t$).

18. The notions of admissibility of an atom, of general example, and of admissible alternatives of a general example are explained in Definition 10-10 to Definition 10-12, respectively.

Note that the methods of all four tasks are fully deterministic (and thus never fail). Moreover, the logic algorithm synthesized at Step 4 is always deterministic.

12.4.3 Correctness

We can now prove the following theorem establishing that Step 4 is in line with the strategy:

Theorem 12-4: *Step* 4 *achieves the two conditions of its objective:*

(1) *$LA_4(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_3(r)$;*

(2) *$exp(LA_4(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_3(r))$.*

Proof 12-4: Let's prove these assertions one by one:

(1) $LA_4(r)$ is trivially equivalent to $LA_3(r)$, hence $LA_4(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_3(r)$.

(2) By assertion (2) of Theorem 7-7 and assertion (2) of Theorem 12-3, it suffices to show that:

$$nonMinimal(X) \wedge decompose(X,\mathbf{HX},\mathbf{TX}) \wedge$$
$$X{=}x_j \wedge Y{=}y_j \wedge \mathbf{HX}{=}\mathbf{hx}_j \wedge \mathbf{TX}{=}\mathbf{tx}_j \Rightarrow \mathbf{r(TX,TY)} \wedge \mathbf{TY}{\in}\mathbf{ty}_j \quad (j{\in}\mathcal{L})$$

in order to prove that $exp(LA_4(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_3(r))$. This is trivially true by the very definition of the *exp*/3 operator, provided a sound oracle is invoked. ❑

12.4.4 Illustration

Let's illustrate Step 4 on the *firstPlateau*/3 and *compress*/2 relations.

Example 12-16: For the *firstPlateau*/3 relation, Step 4 proceeds as follows. The case split is performed, yielding a first version of $LA_4(firstPlateau)$. Task G computes a first version of $exp(LA_4(firstPlateau))$, using the deductive oracle based on $EP(firstPlateau)$. Note that the determinism of *firstPlateau*/3 in the input mode (*ground,any,any*) makes that the oracle finds a singleton set of witnesses to each query. Task H eliminates from the non-recursive case the disjuncts that are expansions wrt the examples E_2 to E_7, because these are also covered by the recursive case. This leaves the expansion of the non-recursive case temporarily empty. Task I removes from the recursive case the disjuncts that are expansions wrt the examples E_3 to E_5, because these do not lead to the existence of admissible alternatives. The non-recursive case is re-expanded wrt these examples. Task J need not eliminate any case. Logic Algorithm 12-7 shows the final version of $LA_4(firstPlateau)$, whereas Logic Algorithm 12-8 shows the final version of $exp(LA_4(firstPlateau))$.

Example 12-17: For the *compress*/2 relation, Step 4 proceeds as follows. The case split is performed, yielding a first version of $LA_4(compress)$. Task G computes a first version of $exp(LA_4(compress))$, using the deductive oracle based on $EP(compress)$. Note that the determinism of *compress*/2 in the input mode (*ground,any*) makes that the oracle finds a singleton set of witnesses to each query. Task H eliminates from the non-recursive case the disjuncts that are expansions wrt the examples E_2 to E_8, because these are also covered by the recursive case. This leaves the expansion of the non-recursive case empty. Task I doesn't remove any disjuncts from the recursive case, because they all lead to the existence of admissible alternatives. Task J eliminates the non-recursive case. The resulting logic algorithm and its expansion are shown in Example 11-1.

```
firstPlateau(L,P,S) ⇔
    L=[_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ firstPlateau(TL,TP,TS)
```

Logic Algorithm 12-7: *LA_4(firstPlateau)*

```
firstPlateau(L,P,S) ⇔
    L=[_]
              ∧     L=[a] ∧ P=[a] ∧ S=[]            {E1}
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧     L=[c,d] ∧ P=[c] ∧ S=[d]
                  ∧ HL=c ∧ TL=[d]                   {E3}
                ∨   L=[e,f,g] ∧ P=[e] ∧ S=[f,g]
                  ∧ HL=e ∧ TL=[f,g]                 {E4}
                ∨   L=[h,i,i] ∧ P=[h] ∧ S=[i,i]
                  ∧ HL=h ∧ TL=[i,i]                 {E5}
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ firstPlateau(TL,TP,TS)
              ∧     L=[b,b] ∧ P=[b,b] ∧ S=[]
                  ∧ HL=b ∧ TL=[b]
                  ∧ TP=[b] ∧ TS=[]                  {E2}
                ∨   L=[j,j,k] ∧ P=[j,j] ∧ S=[k]
                  ∧ HL=j ∧ TL=[j,k]
                  ∧ TP=[j] ∧ TS=[k]                 {E6}
                ∨   L=[m,m,m] ∧ P=[m,m,m] ∧ S=[]
                  ∧ HL=m ∧ TL=[m,m]
                  ∧ TP=[m,m] ∧ TS=[]                {E7}
```

Logic Algorithm 12-8: *exp(LA_4(firstPlateau))*

13 The Reduction Phase

The *reduction phase* of synthesis comprises the last three steps of the synthesis mechanism. In Section 13.1 to Section 13.3, we give complete descriptions of each of these steps: we state their objectives, describe the used methods, analyze their correctness and progression behaviors, and illustrate them on at least the *firstPlateau*/3 and *compress*/2 relations.

13.1 Step 5: Synthesis of *Solve* and the $SolveNonMin_k$

An instantiation of the *Solve* predicate variable computes, in the minimal case, the value of the other parameter *Y* from the induction parameter *X*. An instantiation of the $SolveNonMin_k$ predicate variable computes, in the k^{th} sub-case of the non-recursive case, the value of *Y* from the decomposition ***<HX,TX>*** of *X*.

13.1.1 Objective

The objective at Step 5 is to instantiate the predicate variables *Solve* and $SolveNonMin_k$ of the divide-and-conquer schema. The number *v* of sub-cases of the non-recursive case must also be found. This amounts to transforming $LA_4(r)$ into $LA_5(r)$ such that it is covered by the following schema:

```
r(X,Y) ⇔
            minimal(X)        ∧ Solve(X,Y)
   ∨ ∨1≤k≤v nonMinimal(X)     ∧ decompose(X,HX,TX)
                              ∧ (     SolveNonMink(HX,TX,Y)
                                  ∨
                                      r(TX,TY)                )
```

The following objectives of the strategy:

(1) $LA_5(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_4(r)$;

(2) $exp(LA_5(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_4(r))$;

must be satisfied.

13.1.2 Method

Let's first remember that instances of *Solve*/2 may be defined by fairly complex formulas, including divisions into sub-cases and the corresponding discriminating mechanisms, just as in non-minimal cases. But since this is relatively exceptional, we preferred to keep the schema simple. Rather than excluding such complicated instances of *Solve*/2, we slightly depart from our strict mapping between steps and predicate variables: this Step 5 synthesizes the actual "solving" part of *Solve*/2, while Step 7 synthesizes its "discriminating" part. For this theoretical discussion, we assume however that this Step 5 does the whole job.

A first idea is to use the MSG Method (see Chapter 10). Note that, in both the minimal and the non-recursive case, *Y* may be totally independent of *X*. Indeed, *Y* could be constructed in terms of:

- all the constituents of *X*, as in *LA(efface-L)* (LA 5-5), *LA(insert-R)* (LA 5-9), and in *LA(permutation-L)* (LA 5-14);
- only some of the constituents of *X*, as in *LA(member-L)* (LA 5-10);
- all or some of the constituents of *X*, plus some new constituents, as in *LA(partition-L)* (LA 5-13);
- new constituents only, as in *LA(firstN-N)* (LA 5-7), *LA(parity-L)* (LA 5-12), and in *LA(plateau-N)* (LA 5-15).

The idea is then to invoke the MSG Method twice, so as to tune the admissibility parameters a and b (see Definition 10-10) according to these construction categories. Remember that for admissible examples of arity $a+2b$, the last b parameters are constructed from the median b parameters, while the first a parameters may or may not be used in these constructions. So we first apply the MSG Method on the extracted examples that fall into the first construction category (because the word "all" allows us to fix a=0 and b=1), and we then apply it on all the other extracted examples (where a=2 and b=0).

The synthesis of an instance *solve*/2 of *Solve*/2 may thus be performed by the following sequence of tasks:

- Task K: extract the example set:

  $$\mathcal{E}(solve) = \{\ solve(x_j,y_j)\ \mid\ j\in I \text{ and } x_j, y_j \text{ appear in } exp(LA_4(r))\ \}$$

  from $exp(LA_4(r))$;
- Task L: split $\mathcal{E}$ into two complementary (but possibly empty) sub-sets $\mathcal{E}$' and $\mathcal{E}$", such that every example of $\mathcal{E}$' is admissible wrt a=0 and b=1, and no example of $\mathcal{E}$" is admissible wrt a=0 and b=1. The set $\mathcal{E}$' is relative the minimal examples of r/n where Y is effectively constructed from all the constituents of X, while the set $\mathcal{E}$" is relative the minimal examples of r/n where Y is not constructed from all the constituents of X;
- Task M: invoke the MSG Method (namely the ground case) on the set $\mathcal{E}$' for a=0 and b=1, yielding *LA(solve')*, and invoke the MSG Method (again the ground case) on the set $\mathcal{E}$" for $a=n$=2 and b=0, yielding *LA(solve")*. Admissibility of the provided example sets is achieved in both situations, so the use of the MSG Method is justified. The latter invocation of the MSG Method by construction always discovers a single clique, and may thus possibly miss the appropriate answer;
- Task N: define *LA(solve)* as follows:

```
solve(X,Y) ⇔
          solve'(X,Y)
      v   solve"(X,Y)
```

Similarly, the synthesis of the instances $solveNonMin_k/h+t+1$ of the $SolveNonMin_k/h+t+1$ may be performed by the following sequence of tasks:

- Task O: extract the example set:

  $$\mathcal{E}(solveNonMin) =$$
  $$\{\ solveNonMin(\langle \mathbf{hx}_j,\mathbf{tx}_j\rangle,y_j)\ \mid\ j\in\mathcal{K} \text{ and } \mathbf{hx}_j, \mathbf{tx}_j, y_j \text{ appear in } exp(LA_4(r))\ \}$$

  from $exp(LA_4(r))$;
- Task P: split $\mathcal{E}$ into two complementary (but possibly empty) sub-sets $\mathcal{E}$' and $\mathcal{E}$", such that every example of $\mathcal{E}$' is admissible wrt a=0 and m=1, and no example of $\mathcal{E}$" is admissible wrt a=0 and b=1. The set $\mathcal{E}$' is relative the non-recursive examples of r/n where Y is effectively constructed from all the constituents of the decomposition $\langle \mathbf{HX}, \mathbf{TX}\rangle$ of X, while the set $\mathcal{E}$" is relative the non-recursive examples of r/n where Y is not constructed from all the constituents of $\langle \mathbf{HX},\mathbf{TX}\rangle$;
- Task Q: invoke the MSG Method (namely the ground case) on the set $\mathcal{E}$' for a=0 and b=1, yielding *LA(solveNonMin')*, and invoke the MSG Method (again the ground case) on the set $\mathcal{E}$" for $a=n$=2 and b=0, yielding *LA(solveNonMin")*. Admissibility of the provided example sets is achieved in both situations, so the use of the MSG Method is justified. The latter invocation of the MSG Method by construction always discovers a single clique, and may thus possibly miss the appropriate answer;
- Task R: define *LA(solveNonMin)* as follows:

```
solveNonMin(HX,TX,Y) ⇔
          solveNonMin'(<HX,TX>,Y)
       v  solveNonMin"(<HX,TX>,Y)
```

which may be re-expressed as follows:

```
solveNonMin(HX,TX,Y) ⇔
    v_{1≤k≤v}  solveNonMin_k(HX,TX,Y)
```

hence revealing the value of v, and the instances of the $SolveNonMin_k$.

Every logic algorithm resulting from the MSG Method is here subjected to an assessment heuristic, so as to figure out whether it may be further generalized or not. We have identified the following valuable heuristic for *LA(solve)* (similarly for *LA(solveNonMin)*):

Heuristic 13-1: The msg of a singleton clique is not a real generalization, as it is ground. Let $r(x_i,y_i)$ be the example corresponding to that clique. Some disjunct of *LA(solve)* is thus as follows: $X{=}x_i \wedge Y{=}y_i$. Now, if there is a property $r(s_j,t_j) \Leftarrow B_j$ such that:

(1) $minimal(s_j)$ is provable (where *minimal*/1 is the selected instance of *Minimal*/1),
(2) $r(x_i,y_i) \leq r(s_j,t_j)$ (with substitution σ),
(3) $B_j\sigma$ is provable,

then that disjunct of *LA(solve)* may be semantically generalized (because of (2) and (3)) to: $X{=}s_j \wedge Y{=}t_j \wedge B_j$. This heuristic is justified at the end of Section 13.3.2.

Note that $exp(LA_5(r))$ is as follows:

```
r(X,Y) ⇔
          minimal(X)     ∧ solve(X,Y)
                         ∧      v_{j∈I}     X=x_j ∧ Y=y_j
  v v_{1≤k≤v} nonMinimal(X) ∧ decompose(X,HX,TX)
                         ∧ solveNonMin_k(HX,TX,Y)
                         ∧      v_{j∈K_k}   X=x_j ∧ Y=y_j
                                          ∧ HX=hx_j ∧ TX=tx_j
  v       nonMinimal(X) ∧ decompose(X,HX,TX)
                         ∧ r(TX,TY)
                         ∧      v_{j∈L}     X=x_j ∧ Y=y_j
                                          ∧ HX=hx_j ∧ TX=tx_j
                                          ∧ TY∈ty_j
```

where the $\mathcal{K}_k$ form a partition of $\mathcal{K}$.

The methods of Tasks M and Q are non-deterministic, finite, and never fail, because of the usage of the ground case of the MSG Method. Moreover, the synthesized instances are non-deterministic and finite, for the same reason. The logic algorithm synthesized at Step 5 may thus be non-deterministic, but is certainly finite. All other tasks are fully deterministic. Hence Step 5 never fails.

However, it may happen that the instance of *Solve* or *SolveNonMin* is defined as a full-fledged, possibly recursive, logic algorithm. Different methods need to be applied then. We discuss one such method in Section 14.2.4.

13.1.3 Correctness

We can now prove the following theorem establishing that Step 5 is in line with the strategy:

Theorem 13-1: *Step 5 achieves the two conditions of its objective:*

(1) $LA_5(r)$ *is a better complete approximation of* $\mathcal{R}$ *than* $LA_4(r)$*;*
(2) $exp(LA_5(r))$ *is a better partially correct approximation of* $\mathcal{R}$ *than* $exp(LA_4(r))$*.*

Proof 13-1: Let's prove these assertions one by one:

(1) By assertion (1) of Theorem 7-7 and assertion (1) of Theorem 12-4, it suffices to show that:

$$\mathcal{R}(X,Y) \wedge minimal(X) \Rightarrow solve(X,Y)$$

$$\mathcal{R}(X,Y) \wedge nonMinimal(X) \wedge decompose(X,\mathbf{HX},\mathbf{TX}) \Rightarrow solveNonMin(\mathbf{HX},\mathbf{TX},Y)$$

in order to prove that $LA_5(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_4(r)$. However, the second statement is wrong, so assertion (1) of Theorem 7-7 is not applicable to this disjunct. Instead, we may prove the following statement, obtained by applying assertion (1) of Theorem 7-7 to $LA_3(r)$, which is known by assertion (1) of Theorem 12-3 to be complete wrt $\mathcal{R}$:

$$\mathcal{R}(X,Y) \wedge nonMinimal(X) \wedge decompose(X,\mathbf{HX},\mathbf{TX}) \Rightarrow solveNonMin(\mathbf{HX},\mathbf{TX},Y) \vee \mathbf{r(TX,TY)}$$

The obtained statements are of course unprovable in general, because there is no formal definition of $\mathcal{R}$. But their instances should be proven for each particular synthesis, and they provide thus criteria to be fulfilled by interaction with the specifier.

(2) By assertion (2) of Theorem 7-7 and assertion (2) of Theorem 12-4, it suffices to show that:

$$minimal(X) \wedge X{=}x_j \wedge Y{=}y_j \Rightarrow solve(X,Y) \quad (j \in I)$$

$$nonMinimal(X) \wedge decompose(X,\mathbf{HX},\mathbf{TX}) \wedge X{=}x_j \wedge Y{=}y_j \wedge \mathbf{HX}{=}\mathbf{hx}_j \wedge \mathbf{TX}{=}\mathbf{tx}_j \Rightarrow solveNonMin_k(\mathbf{HX},\mathbf{TX},Y) \quad (1 \le k \le v)\ (j \in \mathcal{K}_k)$$

in order to prove that $exp(LA_5(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_4(r))$. This follows from the correctness of the ground case of the MSG Method (see Theorem 10-3). ❑

This correctness theorem is only applicable to the results obtained before application of the generalization heuristic.

13.1.4 Illustration

Let's illustrate Step 5 on a few relations.

Example 13-1: For the *firstPlateau*/3 relation, Step 5 proceeds as follows. Task K extracts the singleton example set:

```
ℰ = { solveFirstPlateau([a],<[a],[]>) }
```

from the minimal disjunct of $exp(LA_4(firstPlateau))$. At Task L, this example is found to be admissible wrt a=0 and b=1. So $\mathcal{E}' = \mathcal{E}$ and $\mathcal{E}'' = \varnothing$. At Task M, the MSG Method yields:

```
solveFirstPlateau'(L,<P,S>) ⇔
                L=[a] ∧ P=[a] ∧ S=[]
solveFirstPlateau"(L,<P,S>) ⇔
                false
```

The first logic algorithm is based on a singleton clique (originating from *firstPlateau*([*a*],[*a*],[]), that is example E_1 of $\mathcal{E}(firstPlateau)$), and is thus subject to generalization by Heuristic 13-1. Indeed, E_1 is generalized by property P_1 of $\mathcal{P}(firstPlateau)$, so the generalized logic algorithm is:

```
solveFirstPlateau'(L,<P,S>) ⇔
                L=[X] ∧ P=[X] ∧ S=[]
```

At Task N, the instance *solveFirstPlateau*/3 is defined as follows:

```
solveFirstPlateau(L,P,S) ⇔
                P=L ∧ S=[] ∧ L=[_]
```

after performing some obvious simplifications.

Similarly, Task O extracts the example set:

```
E = {  solveNMinFirstPlateau(<c,[d]>,<[c],[d]>)
       solveNMinFirstPlateau(<e,[f,g]>,<[e],[f,g]>)
       solveNMinFirstPlateau(<h,[i,i]>,<[h],[i,i]>)}
```

from the non-recursive disjuncts of *exp(LA*$_4$*(firstPlateau))*. At Task P, these examples are found to be admissible wrt a=0 and b=1. So $\mathcal{E}'=\mathcal{E}$ and $\mathcal{E}''=\varnothing$. At Task Q, the MSG Method yields:

```
solveNMinFirstPlateau'(<HL,TL>,<P,S>) ⇔
                HL=A ∧ TL=[B|T] ∧ P=[A] ∧ S=[B|T]
solveNMinFirstPlateau"(<HL,TL>,<P,S>) ⇔
                false
```

The generalization heuristic doesn't apply here. So, at Task R, the instance *solveNMinFirstPlateau*/4 is defined as follows:

```
solveNMinFirstPlateau(HL,TL,P,S) ⇔
                P=[HL] ∧ S=TL ∧ TL=[_|_]
```

after performing some obvious simplifications.

Hence, Logic Algorithm 13-1 shows *LA*$_5$*(firstPlateau)*, whereas Logic Algorithm 13-2 shows *exp(LA*$_5$*(firstPlateau))*. ♦

Example 13-2: For the *compress*/2 relation, Step 5 proceeds as follows. Task K extracts the singleton example set:

```
E = { solveCompress([],[]) }
```

from the minimal disjunct of *exp(LA*$_4$*(compress))*. At Task L, this example is found to be admissible wrt a=0 and b=1. So $\mathcal{E}'=\mathcal{E}$ and $\mathcal{E}''=\varnothing$. At Task M, the MSG Method yields:

```
solveCompress'(L,C) ⇔
                L=[] ∧ C=[]
solveCompress"(L,C) ⇔
                false
```

The first logic algorithm is based on a singleton clique (originating from *compress*([],[]), that is example E_1 of $\mathcal{E}$*(compress)*), and is thus subject to generalization by Heuristic 13-1. But there is no property in $\mathcal{P}$*(compress)* that generalizes E_1, so no generalization can be performed here. At Task N, the instance *solveCompress*/2 is thus defined as follows:

```
solveCompress(L,C) ⇔
                C=L ∧ L=[]
```

after performing some obvious simplifications. The resulting logic algorithm and its expansion are shown in Example 11-1. ♦

Example 13-3: During the synthesis of *LA(firstN-int-N)*, Step 5 proceeds as follows. Task K extracts the example set:

```
E = { solveFirstN(0,<[], []>)
      solveFirstN(0,<[a],[]>) }
```

from the minimal disjuncts of *exp(LA*$_4$*(firstN))*. At Task L, none of these examples is found to be admissible wrt a=0 and b=1. So $\mathcal{E}''=\mathcal{E}$ and $\mathcal{E}'=\varnothing$. At Task M, the MSG Method yields:

```
firstPlateau(L,P,S) ⇔
    L=[_]      ∧ P=L ∧ S=[] ∧ L=[_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ firstPlateau(TL,TP,TS)
```

Logic Algorithm 13-1: *LA_5(firstPlateau)*

```
firstPlateau(L,P,S) ⇔
    L=[_]      ∧ P=L ∧ S=[] ∧ L=[_]
               ∧     L=[a] ∧ P=[a] ∧ S=[]            {E1}
  ∨ L=[_,_|_] ∧ L=[HL|TL]
               ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
               ∧     L=[c,d] ∧ P=[c] ∧ S=[d]
                   ∧ HL=c ∧ TL=[d]                   {E3}
                 ∨   L=[e,f,g] ∧ P=[e] ∧ S=[f,g]
                   ∧ HL=e ∧ TL=[f,g]                 {E4}
                 ∨   L=[h,i,i] ∧ P=[h] ∧ S=[i,i]
                   ∧ HL=h ∧ TL=[i,i]                 {E5}
  ∨ L=[_,_|_] ∧ L=[HL|TL]
               ∧ firstPlateau(TL,TP,TS)
               ∧     L=[b,b] ∧ P=[b,b] ∧ S=[]
                   ∧ HL=b ∧ TL=[b]
                   ∧ TP=[b] ∧ TS=[]                  {E2}
                 ∨   L=[j,j,k] ∧ P=[j,j] ∧ S=[k]
                   ∧ HL=j ∧ TL=[j,k]
                   ∧ TP=[j] ∧ TS=[k]                 {E6}
                 ∨   L=[m,m,m] ∧ P=[m,m,m] ∧ S=[]
                   ∧ HL=m ∧ TL=[m,m]
                   ∧ TP=[m,m] ∧ TS=[]                {E7}
```

Logic Algorithm 13-2: *exp(LA_5(firstPlateau))*

```
solveFirstN'(N,<L,R>) ⇔
              false
solveFirstN"(N,<L,R>) ⇔
              N=0 ∧ L=T ∧ R=[]
```

At Task N, the instance *solveFirstN*/3 is thus defined as follows:

```
solveFirstN(N,L,R) ⇔
              N=0 ∧ L=_ ∧ R=[]
```

after performing some obvious simplifications. ♦

13.2 Step 6: Synthesis of the $Process_k$ and $Compose_k$

An instantiation of the $Process_k$ predicate variable transforms, in the k^{th} sub-case of the recursive case, the heads ***HX*** of the induction parameter *X* into a vector ***HY*** of heads of the other parameter *Y*. An instantiation of the $Compose_k$ predicate variable computes, in the k^{th} sub-case of the recursive case, the parameter *Y* from its heads ***HY*** (obtained by processing the ***HX***) and tails ***TY*** (obtained by recursion on the ***TX***).

13.2.1 Objective

The objective at Step 6 is to instantiate the predicate variables *Process$_k$* and *Compose$_k$* of the divide-and-conquer schema. The number *w* of sub-cases of the recursive case must also be found. This amounts to transforming *LA$_5$(r)* into *LA$_6$(r)* such that it is covered by the following schema:

```
r(X,Y) ⇔
            minimal(X)       ∧ solve(X,Y)
   ∨ ∨1≤k≤c nonMinimal(X)    ∧ decompose(X,HX,TX))
                             ∧ (     solveNonMin_k(HX,TX,Y)
                                |
                                     r(TX,TY)
                               ∧ Process_k(HX,HY)
                               ∧ Compose_k(HY,TY,Y)       )
```

where $c = v + w$. The following objectives of the strategy:

(1) *LA$_6$(r)* is a better complete approximation of $\mathcal{R}$ than *LA$_5$(r)*;

(2) *exp(LA$_6$(r))* is a better partially correct approximation of $\mathcal{R}$ than *exp(LA$_5$(r))*;

must be satisfied.

13.2.2 Method

We merge the *Process$_k$(**HX**,**HY**)* with the *Compose$_k$(**HY**,**TY**,Y)* into *ProcComp$_k$(**HX**,**TY**,Y)*, so that their instances are synthesized simultaneously. If we isolated the *Process$_k$(**HX**,**HY**)*, then their only universally correct instantiations would be defined as follows:

```
process_k(HX,HY) ⇔ HY=_
```

This means that ***HY*** would have to be initialized to a vector of different variables, and without knowing their number *h'*. Moreover, *w* is yet unknown, so we wouldn't know how many of these instances to create. All this would of course complicate the subsequent synthesis of the *Compose$_k$*. The schema's separation of these predicate variables is thus rather "academic", but not applicable in practice.

We first hypothesize that all instances of the *ProcComp$_k$* are defined in terms of the =/2 primitive only. So we may apply the MSG Method (see Chapter 10). The idea is to initialize the admissibility parameters *a* and *b* (see Definition 10-10) to *h* and 1, respectively, as *Y* (which is of length 1) is necessarily constructed from <***TY***> (which is also of length 1), and optionally from ***HX*** (which is of length *h*). The synthesis of the instances *procComp$_k$*/*h*+*t*+1 of the *ProcComp$_k$*/*h*+*t*+1 may thus be performed by the following sequence of tasks:

- Task S: extract the general example set:

  $$\mathcal{G}(procComp) = \{\ procComp(\mathbf{hx}_j,\langle \mathbf{ty}_j\rangle,y_j) \mid j\in\mathcal{L} \text{ and } \mathbf{hx}_j, \mathbf{ty}_j, y_j \text{ appear in } exp(LA_5(r))\ \}$$

  from *exp(LA$_5$(r))*; [19]
- Task T: invoke the MSG Method (the non-ground case) on the set $\mathcal{G}$ for $a=h$ and $b=1$, yielding *LA(procComp)*. Admissibility of the provided general example set is guaranteed by Step 4 (Syntactic introduction of the recursive atoms), so the use of the MSG Method is justified. The search for admissible alternatives of the general examples may be restricted because, by construction, y_j is ground for every general example in $\mathcal{G}$*(procComp)*, so the minimal hypothesis (see Section 10.4.2) may be emitted;
- Task U: view the synthesized logic algorithm *LA(procComp)* as follows:

19. Note that the extracted general examples are immediately in shorthand form. Also remember (from Step 4) that we do not consider constrained general examples here.

$$\texttt{procComp(}\mathbf{\texttt{HX,<TY>}}\texttt{,Y)} \Leftrightarrow \bigvee_{c-w<k\le v+w} \texttt{procComp}_k\texttt{(}\mathbf{\texttt{HX,TY}}\texttt{,Y)}$$

which reveals the values of w and c, and the instances of the $ProcComp_k$.

Every logic algorithm resulting from the MSG Method is here subjected to some assessment heuristic, so as to figure out whether this method was advisable or not. This assessment is actually performed on the msgs of the discovered cliques, rather than directly on the produced logic algorithm. We have identified the following valuable heuristic:

Heuristic 13-2: If there are at least as many cliques as properties, then the inferred instances probably only cover the given examples, but not examples with "larger" parameters. The assumption of using =/2 only is thus probably too strong.

This heuristic has an obvious influence upon the formulation of a methodology for choosing "good" examples and properties. In the absence of such "good" specifications, a stronger version of this heuristic would be needed. A heuristic similar to Heuristic 13-1 could be easily defined for *LA(procComp)*.

Example 13-4: During the synthesis of *LA(permutation-int-L)*, the MSG Method is invoked as in Example 10-12. The first result is subject to Heuristic 13-2, because there are "too many" cliques, which lets expect that *LA(pcPermutation)* wouldn't behave as intended on a goal such as *pcPermutation(a,[b,c,d,e,f],R)*. The same holds for the second result. The intended relation for *LA(pcPermutation)* is indeed the one underlying the *stuff*/3 predicate, and thus requires a recursive logic algorithm. We show in Section 14.2.4 how this can be obtained.

Note that $exp(LA_6(r))$ is as follows:

$$\begin{aligned}
\texttt{r(X,Y)} \Leftrightarrow\ & \texttt{minimal(X)} \wedge \texttt{solve(X,Y)} \\
& \wedge \bigvee_{j\in I} \texttt{X=x}_j \wedge \texttt{Y=y}_j \\
\vee\ & \texttt{nonMinimal(X)} \wedge \texttt{decompose(X,}\mathbf{\texttt{HX,TX}}\texttt{)} \\
& \wedge \texttt{solveNonMin}_k\texttt{(}\mathbf{\texttt{HX,TX}}\texttt{,Y)} \\
& \wedge \bigvee_{j\in\mathcal{K}_k} \texttt{X=x}_j \wedge \texttt{Y=y}_j \wedge \mathbf{\texttt{HX=hx}}_j \wedge \mathbf{\texttt{TX=tx}}_j \\
\vee\ & \texttt{nonMinimal(X)} \wedge \texttt{decompose(X,}\mathbf{\texttt{HX,TX}}\texttt{)} \\
& \wedge \mathbf{\texttt{r(TX,TY)}} \\
& \wedge \texttt{procComp}_k\texttt{(}\mathbf{\texttt{HX,<TY>}}\texttt{,Y)} \\
& \wedge \bigvee_{j\in\mathcal{L}_k} \texttt{X=x}_j \wedge \texttt{Y=y}_j \wedge \mathbf{\texttt{HX=hx}}_j \wedge \mathbf{\texttt{TX=tx}}_j \wedge \mathbf{\texttt{TY}\in\texttt{ty'}}_j
\end{aligned}$$

where the $\mathcal{L}_k$ form a partition of $\mathcal{L}$, and where the elements ty'_{jk} of the $\mathbf{ty'}_j$ are subsets of the sets of witnesses ty_{jk} generated at Step 4 (Syntactic introduction of the recursive atoms).

Example 13-5: Let's reconsider Example 7-13, but with *LA(permutation)* now as follows:

$$\begin{aligned}
&\texttt{permutation(L,P)} \Leftrightarrow \\
&\quad \texttt{L=[]} \\
&\vee\ \texttt{L=[HL|TL]} \wedge \texttt{permutation(TL,TP)} \\
&\qquad\qquad \wedge \texttt{efface(HL,P,TP)}
\end{aligned}$$

The expansion of the second disjunct of *LA(permutation)* wrt *permutation*([*a*,*b*,*c*],[*c*,*a*,*b*]) and the deductive oracle based on *EP(permutation)* is:

$$L{=}[HL|TL] \wedge permutation(TL,TP) \wedge efface(HL,P,TP) \wedge (L{=}[a,b,c] \wedge P{=}[c,a,b]) \wedge (HL{=}a \wedge TL{=}[b,c] \wedge TP{=}[c,b])$$

Indeed, if *TL*=[*b*,*c*], then the deductive oracle infers that either *TP*=[*b*,*c*] or *TP*=[*c*,*b*], according to properties P_2 and P_3, respectively. But the *efface*/3 atom only holds for the first answer. ♦

If the result of the MSG Method is rejected (due to Heuristic 13-2), then it is likely that the instance of *ProcComp* is defined as a full-fledged, possibly recursive, logic algorithm. Different methods need to be applied then. We discuss one such method in Section 14.2.4.

Note that the method of Task T is non-deterministic, always succeeds, but may be infinite, because of the non-ground case of the MSG Method. Moreover, the synthesized instances are non-deterministic and finite, for the same reason. The logic algorithm synthesized at Step 6 may thus be non-deterministic, but is certainly finite. All other tasks are fully deterministic. Hence Step 6 never fails.

As the ground case of the MSG Method is much easier, this speaks in retrospect in favor of using the Functionality Heuristic when selecting an induction parameter at Step 2. Indeed, this forces all the $\boldsymbol{ty}_j$ to be ground and unique, so that all the extracted general examples are mere (ground) examples, because all the $\boldsymbol{hx}_j$ and y_j are necessarily ground and unique. This is the best possible situation. The worst possible situation for using the MSG Method is the non-ground case where the parameters of each general example are existential variables. This cannot happen here, because all the $\boldsymbol{hx}_j$ and y_j are necessarily ground and unique. So the worst situation at Step 6 would be the one where the *exp*/3 operator generates a variable for each element of each $\boldsymbol{ty}_j$, because of some "dumb" oracle for *r*/*n*. The MSG Method can still handle such a worst-case situation, as it then enumerates all possible grounding substitutions in its search for admissible alternatives of each general example. This may be very time-consuming of course. The point however is that the oracle used by the *exp*/3 operator is irrelevant, provided it is sound.

13.2.3 Correctness

We can now prove the following theorem establishing that Step 6 is in line with the strategy:

Theorem 13-2: *Step* 6 *achieves the two conditions of its objective:*

(1) *$LA_6(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_5(r)$;*

(2) *$exp(LA_6(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_5(r))$.*

Proof 13-2: Let's prove these assertions one by one:

(1) By assertion (1) of Theorem 7-7 and assertion (1) of Theorem 13-1, it suffices to show that:

$$\mathcal{R}(X,Y) \wedge nonMinimal(X) \wedge decompose(X,\boldsymbol{HX},\boldsymbol{TX}) \wedge \boldsymbol{r(TX,TY)} \Rightarrow procComp(\boldsymbol{HX},\boldsymbol{TY},Y)$$

in order to prove that $LA_6(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_5(r)$. However, this statement is wrong, so assertion (1) of Theorem 7-7 is not applicable to this disjunct of $LA_5(r)$. Instead, we may prove the following statement, obtained by applying assertion (1) of Theorem 7-7 to $LA_3(r)$, which is known by assertion (1) of Theorem 12-3 to be complete wrt $\mathcal{R}$:

$$\mathcal{R}(X,Y) \wedge nonMinimal(X) \wedge decompose(X,\boldsymbol{HX},\boldsymbol{TX}) \Rightarrow solveNonMin(\boldsymbol{HX},\boldsymbol{TX},Y) \vee (\boldsymbol{r(TX,TY)} \wedge procComp(\boldsymbol{HX},\boldsymbol{TY},Y))$$

This is of course unprovable in general, because there is no formal definition of $\mathcal{R}$. But the corresponding instance should be proven for each particular synthesis, and this provides thus a criterion to be fulfilled by interaction with the specifier.

(2) By assertion (2) of Theorem 7-7 and assertion (2) of Theorem 13-1, it suffices to show that:

$$nonMinimal(X) \wedge decompose(X,\mathbf{HX},\mathbf{TX}) \wedge \mathbf{r(TX,TY)} \wedge$$
$$X=x_j \wedge Y=y_j \wedge \mathbf{HX}=\mathbf{hx}_j \wedge \mathbf{TX}=\mathbf{tx}_j \wedge \mathbf{TY}\in\mathbf{ty'}_j \Rightarrow$$
$$procComp_k(\mathbf{HX},\mathbf{TY},Y) \quad (c-w<k\leq c)\ (j\in\mathcal{L}_k)$$

in order to prove that $exp(LA_6(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_5(r))$. This follows from the correctness of the non-ground case of the MSG Method (see Theorem 10-4). ❑

This correctness theorem is only applicable to the results obtained before application of the generalization heuristic.

13.2.4 Illustration

Let's illustrate Step 6 on a few relations.

Example 13-6: For the *firstPlateau*/3 relation, Step 6 proceeds as in Example 10-7. The resulting logic algorithm is subject to no heuristic. Logic Algorithm 13-3 shows $LA_6(firstPlateau)$, whereas Logic Algorithm 13-4 shows $exp(LA_6(firstPlateau))$.

Example 13-7: For the *compress*/2 relation, Step 6 proceeds as in Example 10-6. The obtained result is subject to no heuristic. The resulting logic algorithm and its expansion are shown in Example 11-1.

Example 13-8: During the synthesis of *LA(plateau-int-N)*, the MSG Method is invoked as in Example 10-11. The resulting logic algorithm is subject to no heuristic.

13.3 Step 7: Synthesis of the $Discriminate_k$

An instantiation of the $Discriminate_k$ predicate variable tests the values of $\mathbf{HX}$, $\mathbf{TX}$, and Y in order to see whether $SolveNonMin_k$, respectively $Process_k \wedge Compose_k$, is applicable.

13.3.1 Objective

The objective at Step 7 is to instantiate the predicate variables $Discriminate_k$ of the divide-and-conquer schema. This amounts to transforming $LA_6(r)$ into $LA_7(r)$ such that it is covered by the following schema:

```
r(X,Y) ⇔
          minimal(X)      ∧ solve(X,Y)
   ∨ ∨1≤k≤c nonMinimal(X) ∧ decompose(X,HX,TX)
                          ∧ Discriminatek(HX,TX,Y)
                          ∧ (     solveNonMink(HX,TX,Y)
                              |
                                  r(TX,TY)
                            ∧ processk(HX,HY)
                            ∧ composek(HY,TY,Y)       )
```

The following objectives of the strategy:

(1) $LA_7(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_6(r)$;

(2) $exp(LA_7(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_6(r))$;

must be satisfied.

Also, Step 7 should be totally independent of the choice of Step 6 to merge the $process_k$ and $compose_k$ predicates.

```
firstPlateau(L,P,S) ⇔
    L=[_]      ∧ P=L ∧ S=[] ∧ L=[_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ firstPlateau(TL,TP,TS)
              ∧ P=[HL|TP] ∧ S=TS ∧ TP=[HL|_]
```

Logic Algorithm 13-3: *LA$_6$(firstPlateau)*

```
firstPlateau(L,P,S) ⇔
    L=[_]      ∧ P=L ∧ S=[] ∧ L=[_]
               ∧     L=[a] ∧ P=[a] ∧ S=[]            {E1}
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
              ∧     L=[c,d] ∧ P=[c] ∧ S=[d]
                  ∧ HL=c ∧ TL=[d]                   {E3}
                ∨   L=[e,f,g] ∧ P=[e] ∧ S=[f,g]
                  ∧ HL=e ∧ TL=[f,g]                 {E4}
                ∨   L=[h,i,i] ∧ P=[h] ∧ S=[i,i]
                  ∧ HL=h ∧ TL=[i,i]                 {E5}
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ firstPlateau(TL,TP,TS)
              ∧ P=[HL|TP] ∧ S=TS ∧ TP=[HL|_]
              ∧     L=[b,b] ∧ P=[b,b] ∧ S=[]
                  ∧ HL=b ∧ TL=[b]
                  ∧ TP=[b] ∧ TS=[]                  {E2}
                ∨   L=[j,j,k] ∧ P=[j,j] ∧ S=[k]
                  ∧ HL=j ∧ TL=[j,k]
                  ∧ TP=[j] ∧ TS=[k]                 {E6}
                ∨   L=[m,m,m] ∧ P=[m,m,m] ∧ S=[]
                  ∧ HL=m ∧ TL=[m,m]
                  ∧ TP=[m,m] ∧ TS=[]                {E7}
```

Logic Algorithm 13-4: *exp(LA$_6$(firstPlateau))*

13.3.2 Method

The bodies of the properties contain explicit information that has not yet been synthesized into the logic algorithms. But this explicit knowledge constitutes the very reason why we augmented example-based specifications with properties. The idea is then to use the Proofs-as-Programs Method (see Chapter 9), as this amounts to "transferring" appropriate variants of atoms from the bodies of properties to the body of a logic algorithm. We thus invoke the Proofs-as-Programs Method with the following *input*:

- use *LA$_6$(r)* as the input logic algorithm;
- use *EP(r)* as the input property set (remember that examples are properties, too);

Indeed, the pre-condition is achieved because *LA$_6$(r)* only uses, by Steps 1 to 6, primitive predicates and the *r/n* predicate, and is, by Theorem 13-2, complete wrt *EP(r)*.

- use the following set $\mathcal{H}$ of generalization heuristics:

Heuristic 13-3: From the used divide-and-conquer schema, we know that the parameters of the $discriminate_k$ are ***HX***, ***TX***, and *Y*, so we should *project* [Pettorossi and Proietti 89] the discriminants upon these parameters.

Heuristic 13-4: If parameter *Y* is not an auxiliary parameter (see Section 14.2.2 on how to detect auxiliary parameters), then it is irrelevant for discrimination, and its equality atom may be deleted from the body of every discriminant.

Heuristic 13-5: The parameters ***TX*** and *Y* (the latter only if it is an auxiliary parameter) should range across their entire domains: if necessary, some of their values should be generalized. In our case, this goes as follows:

- if TX_i is an integer and the atom $TX_i = s(s(\ldots s(0)\ldots))$ appears in the last disjunct of a discriminant, then generalize that atom to $TX_i = s(s(\ldots s(N)\ldots))$, where *N* is a new variable; also delete the equality atom involving HX_i;
- if TX_i is a list and the atom $TX_i = [\boldsymbol{s}]$ appears in the last disjunct of a discriminant, then generalize that atom to $TX_i = [\boldsymbol{s}|T]$, where ***s*** are terms, and *T* is a new variable.

The generalization of *Y* is similar, provided, of course, it is of an inductive type.

The application of the last two heuristics should be interactive and in accordance with the given examples.

The *output* is $LA_7(r)$, and the newly added atoms (compared to $LA_6(r)$) constitute the instances of the $Discriminate_k$ predicates. Note that $exp(LA_7(r))$ is as follows:

$$
\begin{array}{lll}
\texttt{r(X,Y)} \Leftrightarrow & & \\
 & \texttt{minimal(X)} & \wedge\ \texttt{solve(X,Y)} \\
 & & \wedge \quad \vee_{j\in I} \quad \texttt{X=}x_j \wedge \texttt{Y=}y_j \\
\vee \vee_{1\le k\le v} & \texttt{nonMinimal(X)} & \wedge\ \texttt{decompose(X,}\mathbf{HX}\texttt{,}\mathbf{TX}\texttt{)} \\
 & & \wedge\ \texttt{discriminate}_k\texttt{(}\mathbf{HX}\texttt{,}\mathbf{TX}\texttt{,Y)} \\
 & & \wedge\ \texttt{solveNonMin}_k\texttt{(}\mathbf{HX}\texttt{,}\mathbf{TX}\texttt{,Y)} \\
 & & \wedge \quad \vee_{j\in \mathcal{K}_k} \quad \texttt{X=}x_j \wedge \texttt{Y=}y_j \\
 & & \qquad\qquad \wedge\ \mathbf{HX}\texttt{=}\mathbf{hx}_j \wedge \mathbf{TX}\texttt{=}\mathbf{tx}_j \\
\vee \vee_{c-w<k\le c} & \texttt{nonMinimal(X)} & \wedge\ \texttt{decompose(X,}\mathbf{HX}\texttt{,}\mathbf{TX}\texttt{)} \\
 & & \wedge\ \texttt{discriminate}_k\texttt{(}\mathbf{HX}\texttt{,}\mathbf{TX}\texttt{,Y)} \\
 & & \wedge\ \mathbf{r(TX,TY)} \\
 & & \wedge\ \texttt{procComp}_k\texttt{(}\mathbf{HX}\texttt{,}\mathbf{TY}\texttt{,Y)} \\
 & & \wedge \quad \vee_{j\in \mathcal{L}_k} \quad \texttt{X=}x_j \wedge \texttt{Y=}y_j \\
 & & \qquad\qquad \wedge\ \mathbf{HX}\texttt{=}\mathbf{hx}_j \wedge \mathbf{TX}\texttt{=}\mathbf{tx}_j \\
 & & \qquad\qquad \wedge\ \mathbf{TY}\in\mathbf{ty'}_j
\end{array}
$$

As a reminder, Step 7 is also supposed to synthesize the "discriminating" part of *Solve*/2. But we again assume that the instance *solve*/2 found by Step 5 is left unchanged by this Step 7.

Note that the Proofs-as-Programs Method wouldn't need to prove that the examples are logical consequences of the input logic algorithm, as we know that $LA_6(r)$ is already complete wrt $\mathcal{E}(r)$. Moreover, such proofs would result in very specialized discriminants, and hence in the need for considerable generalization afterwards. But the presence of some examples is often necessary within the theories, so that DCI may reduce atoms of predicate *r*/*n* in case no property is applicable. The following definition and heuristic capture this:

Definition 13-1: Given a specification and a minimal form, a *property-example* is a minimal example that is a logical consequence of no property.

Example 13-9: The following are property-examples: *compress*([],[]) and *length*([],0). But *plateau*(1,*b*,[*b*]) and *firstN*(0,[],[]) are not property-examples, because they are logical consequences of the first properties of their respective specifications.

Heuristic 13-6: At Step 7, only provide the properties and property-examples of *EP(r)* as input property set to the Proofs-as-Programs Method.

Step 7 is deterministic, because of the Proofs-as-Programs Method. It fails iff $LA_6(r)$ is not complete wrt $\mathcal{P}(r)$. This should normally be excluded by the conjunction of assertion (1) of Theorem 13-2 and the consistency assumption. Moreover, nothing can be said about the synthesized discriminants in terms of determinism or finiteness, because nothing is known about the properties.

We may now justify Heuristic 13-1. When proving the involved property, say P_j, the derivation starting from the clause, say C_i, that is obtained from the generalized minimal disjunct necessarily succeeds, as shown in the following derivation schema:

$$\mathbf{r(s_j,t_j)} \leftarrow B_j$$

$$DCI\text{: } C_i \quad \downarrow \quad \{\}$$

$$\mathbf{minimal(s_j)} \ \&\ s_j{=}s_j \ \&\ t_j{=}t_j \ \&\ B_j \leftarrow B_j$$

$$DCI\text{: } LA(minimal) \text{ and constraint (1)} \quad \downarrow \quad \{\}$$

$$\mathbf{s_j{=}s_j \ \&\ t_j{=}t_j \ \&\ B_j \leftarrow B_j}$$

$$2 \times DCI, Sim\text{: } LA(=) \quad \downarrow \quad \{\}$$

❑

Property P_j is thus a logical consequence of the synthesized logic algorithm: the performed generalization is thus legal. Note that the extracted discriminant is redundant with the existing literals of the involved disjunct. If the heuristic hadn't already added the conjunction B_j to that disjunct, then the extracted discriminant would have been B_j.

13.3.3 Correctness

We can now prove the following theorem establishing that Step 7 is in line with the strategy:

Theorem 13-3: *Step 7 achieves the two conditions of its objective:*

(1) *$LA_7(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_6(r)$;*

(2) *$exp(LA_7(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_6(r))$.*

Proof 13-3: Let's prove these assertions one by one:

(1) By assertion (1) of Theorem 7-7 and assertion (1) of Theorem 13-2, it suffices to show that:

$$\mathcal{R}(X,Y) \wedge nonMinimal(X) \wedge decompose(X,\mathbf{HX},\mathbf{TX}) \wedge solveNonMin_k(\mathbf{HX},\mathbf{TX},Y) \Rightarrow discriminate_k(\mathbf{HX},\mathbf{TX},Y) \quad (1 \le k \le v)$$

$$\mathcal{R}(X,Y) \wedge nonMinimal(X) \wedge decompose(X,\mathbf{HX},\mathbf{TX}) \wedge \mathbf{r(TX,TY)} \wedge procComp_k(\mathbf{HX},\mathbf{TY},Y) \Rightarrow discriminate_k(\mathbf{HX},\mathbf{TX},Y) \quad (c-w < k \le c)$$

in order to prove that $LA_7(r)$ is a better complete approximation of $\mathcal{R}$ than $LA_6(r)$. This is of course unprovable in general, because there is no formal definition of $\mathcal{R}$. But the corresponding instance should be proven for each particular synthesis, and this provides thus a criterion to be fulfilled by interaction with the specifier.

(2) By assertion (2) of Theorem 7-7 and assertion (2) of Theorem 13-2, it suffices to show that:

$$nonMinimal(X) \wedge decompose(X,\mathbf{HX},\mathbf{TX}) \wedge solveNonMin_k(\mathbf{HX},\mathbf{TX},Y) \wedge X{=}x_j \wedge Y{=}y_j \wedge \mathbf{HX}{=}\mathbf{hx}_j \wedge \mathbf{TX}{=}\mathbf{tx}_j \Rightarrow discriminate_k(\mathbf{HX},\mathbf{TX},Y) \quad (1 \le k \le v)\ (j \in \mathcal{K}_k)$$

$$nonMinimal(X) \wedge decompose(X,\mathbf{HX},\mathbf{TX}) \wedge \mathbf{r(TX,TY)} \wedge procComp_k(\mathbf{HX},\mathbf{TY},Y) \wedge$$
$$X=x_j \wedge Y=y_j \wedge \mathbf{HX}=\mathbf{hx}_j \wedge \mathbf{TX}=\mathbf{tx}_j \wedge \mathbf{TY} \in \mathbf{ty'}_j \Rightarrow$$
$$discriminate_k(\mathbf{HX},\mathbf{TX},Y) \quad (c-w<k\leq c)\ (j\in\mathcal{L}_k)$$

in order to prove that $exp(LA_7(r))$ is a better partially correct approximation of $\mathcal{R}$ than $exp(LA_6(r))$. This follows from the correctness of the Proofs-as-Programs Method (see Theorem 9-2). ❑

This correctness theorem is only applicable to the results obtained before application of the generalization heuristics.

13.3.4 Illustration

Let's illustrate Step 7 on the *firstPlateau*/3 and *compress*/2 relations.

Example 13-10: For the *firstPlateau*/3 relation, Step 7 proceeds as in Example 9-5. There are no property-examples, so Heuristic 13-6 only provides the properties of *EP(firstPlateau)* as a specification to the Proofs-as-Programs Method. The applied generalization heuristics are Heuristic 13-3, Heuristic 13-4, and Heuristic 13-5, respectively. Logic Algorithm 13-5 shows LA_7*(firstPlateau)*, whereas Logic Algorithm 13-6 shows $exp(LA_7$*(firstPlateau))*.

Example 13-11: For the *compress*/2 relation, Step 7 proceeds as in Example 9-4. There is one property-example, namely E_1, so Heuristic 13-6 provides E_1 and the properties of *EP(compress)* as a specification to the Proofs-as-Programs Method. The applied generalization heuristics are Heuristic 13-3, Heuristic 13-4, and Heuristic 13-5, respectively. The resulting logic algorithm and its expansion are shown in Example 11-1.

```
firstPlateau(L,P,S) ⇔
    L=[_]      ∧ P=L ∧ S=[] ∧ L=[_] ∧ L=[_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ TL=[HTL|_] ∧ HL≠HTL
              ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
  ∨ L=[_,_|_] ∧ L=[HL|TL]
              ∧ TL=[HTL|_] ∧ HL=HTL
              ∧ firstPlateau(TL,TP,TS)
              ∧ P=[HL|TP] ∧ S=TS ∧ TP=[HL|_]
```

Logic Algorithm 13-5: *LA$_7$(firstPlateau)*

```
firstPlateau(L,P,S) ⇔
    L=[_]      ∧ P=L ∧ S=[] ∧ L=[_] ∧ L=[_]
               ∧     L=[a] ∧ P=[a] ∧ S=[]            {E1}
  ∨ L=[_,_|_] ∧ L=[HL|TL]
               ∧ TL=[HTL|_] ∧ HL≠HTL
               ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
               ∧     L=[c,d] ∧ P=[c] ∧ S=[d]
                   ∧ HL=c ∧ TL=[d]                  {E3}
                 ∨   L=[e,f,g] ∧ P=[e] ∧ S=[f,g]
                   ∧ HL=e ∧ TL=[f,g]                {E4}
                 ∨   L=[h,i,i] ∧ P=[h] ∧ S=[i,i]
                   ∧ HL=h ∧ TL=[i,i]                {E5}
  ∨ L=[_,_|_] ∧ L=[HL|TL]
               ∧ TL=[HTL|_] ∧ HL=HTL
               ∧ firstPlateau(TL,TP,TS)
               ∧ P=[HL|TP] ∧ S=TS ∧ TP=[HL|_]
               ∧     L=[b,b] ∧ P=[b,b] ∧ S=[]
                   ∧ HL=b ∧ TL=[b]
                   ∧ TP=[b] ∧ TS=[]                 {E2}
                 ∨   L=[j,j,k] ∧ P=[j,j] ∧ S=[k]
                   ∧ HL=j ∧ TL=[j,k]
                   ∧ TP=[j] ∧ TS=[k]                {E6}
                 ∨   L=[m,m,m] ∧ P=[m,m,m] ∧ S=[]
                   ∧ HL=m ∧ TL=[m,m]
                   ∧ TP=[m,m] ∧ TS=[]               {E7}
```

Logic Algorithm 13-6: *exp(LA$_7$(firstPlateau))*

14 Discussion

In this chapter, we draw some conclusions about the synthesis mechanism presented in the previous two chapters. First, in Section 14.1, we evaluate its current form wrt the objectives. Then, in Section 14.2, we imagine some extensions and outline some future work. In Section 14.3, we outline a methodology for choosing "good" examples and properties. A prototype implementation of the synthesis mechanism is being developed: it is called SYNAPSE (*SYNthesis of logic Algorithms from PropertieS and Examples*), and is written in portable Prolog. We discuss its architecture and illustrate its working on some target synthesis scenarios in Section 14.4. Finally, in Section 14.5, we evaluate our synthesis mechanism wrt to some related mechanisms.

14.1 Summary

We have described a logic algorithm synthesis mechanism that is stepwise, guided by a divide-and-conquer schema, non-incremental, monotonically decreasing and increasing, as well as consistent. We conjecture that the logic algorithm synthesized at Step 7, that is $LA_7(r)$, is totally correct wrt the intended relation $\mathcal{R}$. Reliability can however not be guaranteed. Moreover, the synthesis mechanism is highly algorithmic and potentially interactive. We have no theoretical results about its power or data-efficiency.

Domain generality and full automation have been sacrificed for the sake of better end-user orientation, and because this is a most reasonable approach with incomplete specifications (where one deliberately admits a gap between the specified relation and the intended relation). Incorporated knowledge sources are the algorithms knowledge of the divide-and-conquer schema, and the domain knowledge of the typed databases. There is no meta-knowledge, and no efficiency knowledge. Whole algorithm families can be synthesized from a single specification, because some synthesis steps are non-deterministic.

Table 14-1 charts a summary of the seven synthesis steps and of their sub-tasks. The columns list (from left to right):

- the name(s) of the instantiated (predicate) variable(s) of version 3 of the divide-and-conquer schema;
- the used method(s) of the tool-box (where MSG_1 stands for the ground case of the MSG Method, MSG_2 stands for the non-ground case of the MSG Method, and *P-as-P* stands for the Proofs-as-Programs Method);
- the determinism (minimal and maximal number of possible different answers) of the method underlying each step or task, without the usage of any heuristics; where applicable, a footnote explains under which condition there are no answers, or in what way the heuristics may alter this determinism;
- the time-complexity of the method underlying each step or task, again without the usage of any heuristics;
- the page where the method underlying each step or task is explained.

Empty cells denote non-applicability of a criterion wrt a step or task.

This table shows that all the (predicate) variables of version 3 of the divide-and-conquer schema are instantiated by the synthesis mechanism, using all the methods of the tool-box developed in Part II to do so. Overall, synthesis may fail (for a variety of individual reasons), but it may on the other hand succeed an infinite number of times, yielding as many logic algorithms. The overall mechanism is NP-complete. This is however not dramatic as the number m of examples usually is quite small.

Table 14-1: Summary of the synthesis steps and tasks

		Variable(s)	Method	Answers	Complexity [a]	Page
Step 1				1 – 1	$O(m)$	166
Step 2	Task A			0 – n [b]	$O(n)$	168
	Task B			1 – 1	$O(m \log_2 m)$	168
	Task C			1 – m	$O(m)$	169
	Task D	*Minimal*, *NonMinimal*	Database	1 – ∗	$O(1)$	169
Step 3	Task E			1 – ∗	$O(1)$	172
	Task F	*Decompose*	Database	1 – ∗	$O(1)$	173
Step 4	Task G			1 – 1	$O(t\,(m+p))$	177
	Task H			1 – 1	$O(m)$	177
	Task I			1 – 1	$O(m)$	177
	Task J			1 – 1	$O(1)$	177
Step 5	Task K Task O			1 – 1 1 – 1	$O(m)$ $O(m)$	182 182
	Task L Task P			1 – 1 1 – 1	$O(m)$ $O(m)$	182 182
	Task M Task Q		MSG_1 MSG_1	1 – ∗ [c] 1 – ∗ [d]	NP NP	182 182
	Task N Task R	*Solve*, *v*, $SolveNonMin_k$		1 – 1 1 – 1	$O(1)$ $O(1)$	182 182
Step 6	Task S			1 – 1	$O(m)$	187
	Task T		MSG_2	1 – ∞ [e]	NP	187
	Task U	*w*, *c*, $Process_k$, $Compose_k$		1 – 1	$O(1)$	187
Step 7		$Discriminate_k$	P-as-P	0 – 1 [f]	$O(cp^{t+1})$	191
Steps 1 to 7		All variables	Tool-box	0 – ∞	NP	

a. Where m is the number of examples, p is the number of properties, n is the arity of predicate r, t is the number of recursive atoms, and c is the number of non-minimal cases. NP denotes an NP-complete problem.
b. No answers if there is no parameter of an inductive type.
c. No answers if some heuristic rejects all the results of the MSG Method.
d. No answers if some heuristic rejects all the results of the MSG Method.
e. No answers if Heuristic 13-2 rejects all the results of the MSG Method.
f. No answers if $LA_6(r)$ is not complete wrt $\mathcal{P}(r)$.

Table 14-2 charts a summary of the features of the logic algorithms synthesized at step i. The columns list (from left to right):

- the determinism of the added atoms, given a ground value of the induction parameter, the other parameters being variables;
- the determinism of the synthesized logic algorithm $LA_i(r)$, given a ground value of the induction parameter;
- the characterization of $LA_i(r)$ in terms of correctness and progression; where applicable, a footnote explains under which condition this behavior is achieved;
- the usage (denoted *yes*) or non-usage (denoted *no*) of the property set $\mathcal{P}(r)$;
- the number of cases of $LA_i(r)$, represented as a sum $u+v+w$, where u is the number of minimal cases, v is the number of non-recursive cases, and w is the number of recursive cases.

This table shows that the instances of the predicate variables may be defined by algorithms of any kind of determinism. The synthesized logic algorithms $LA_i(r)$ evolve from full determinism ($i = 1$) to determinism ($i = 2\ldots4$) to potential non-determinism ($i = 5\ldots7$), but are always terminating. The logic algorithms $exp(LA_i(r))$ exhibit a non-strict upward progression, starting from a logic algorithm that is totally correct wrt the example set. On the other hand, the logic algorithms $LA_i(r)$ exhibit a possibly strict downward progression, starting from the top logic algorithm. The property set is used by Step 4 (Syntactic introduction of the recursive atoms) and Step 7 (Synthesis of the discriminants). The number of cases of the $LA_i(r)$ evolves from 1 ($i = 1$) to 2 ($i = 2\ldots3$) to 3 ($i = 3\ldots4$) to a sum $1+v+w$ ($i = 5\ldots7$), where v and w are any integers. Of course, if it turns out that w=0, then this synthesis mechanism is inadequate.

Table 14-2: Summary of the features of the synthesized logic algorithms

	New atoms	$LA_i(r)$	Correctness and Progression	$\mathcal{P}$(r)	Cases
Step 1	N/A	1 – 1	$\mathcal{E} \cong exp(LA_1) \leq \mathcal{R} \leq LA_1 = \bot$ [a]	no	1
Step 2	0 – 1 0 – 1	0 – 1	$exp(LA_1) \cong exp(LA_2) \leq \mathcal{R} \leq LA_2 \leq LA_1$ [b]	no no	1+1
Step 3	0 – 1	0 – 1	$exp(LA_2) \cong exp(LA_3) \leq \mathcal{R} \leq LA_3 \leq LA_2$	no	1+1
Step 4	N/A	0 – 1	$exp(LA_3) \cong exp(LA_4) \leq \mathcal{R} \leq LA_4 \cong LA_3$ [c]	yes	1+1+1
Step 5	0 – * 0 – *	0 – *	$exp(LA_4) \cong exp(LA_5) \leq \mathcal{R} \leq LA_5 \leq LA_4$	no no	1+1+1 1+v+1
Step 6	0 – *	0 – *	$exp(LA_5) \cong exp(LA_6) \leq \mathcal{R} \leq LA_6 \leq LA_5$	no	1+v+w
Step 7	? – ?	0 – *	$exp(LA_6) \cong exp(LA_7) \leq \mathcal{R} \leq LA_7 \leq LA_6$	yes	1+v+w
Steps 1 to 7	N/A	0 – *	conjecture: $\mathcal{R} \cong LA = LA_7$	yes	1+v+w

a. Under the consistency assumption.
b. If the domain extrapolation hypothesis is correct.
c. If a sound oracle $O(r)$ is used.

We are now in position to analyze which of the challenges of Section 5.3 have been met, and to state how they have been met, respectively why they have not been met:

- *What induction parameter to select?* Task A of Step 2 (Synthesis of *Minimal* and *Non-Minimal*) does this non-deterministically by considering all parameters that are of an inductive type, and possibly by using some selection heuristics, or by listening to the specifier's hints.
- *How to discover compound induction parameters?* Due to our restriction to version 3 of the divide-and-conquer schema, Task A only considers simple induction parameters. Meeting this challenge is thus considered future research, but shouldn't be too difficult.
- *According to what well-founded relation to decompose the induction parameter?* Step 3 (Synthesis of *Decompose*) does this non-deterministically by considering all predefined decomposition operators (which each reflect some well-founded relation) of a typed database, and possibly by listening to the specifier's hints.
- *Is the mechanism able to design the whole family of possible logic algorithms for a given problem?* This ability is in theory achieved for the family of logic algorithms that are covered by version 3 of the divide-and-conquer schema. In practice, everything depends on the completeness of the typed databases used by Steps 2 and 3.
- *How many structural forms are there?* Due to our restriction to version 3 of the divide-and-conquer schema, Task C of Step 2 only considers the distinction between 2 structural forms, namely one minimal form and one non-minimal form. A generalization of this task is considered future research.
- *What are the structural forms?* Task D of Step 2 non-deterministically selects some domain extrapolation hypothesis and extracts the corresponding predefined forms from a typed database.
- *Into how many cases is each structural case divided?* The progressive discovery of this number is summarized in the last column of Table 14-2.
- *How to discriminate between these sub-cases?* Step 7 (Synthesis of the $Discriminate_k$) achieves this by using the Proofs-as-Programs Method.
- *How to detect that recursion is useless in some non-minimal sub-cases?* Step 4 (Syntactic introduction of the recursive atoms) creates a non-recursive non-minimal case if at least one example E_j leads to a non-admissible $procComp(\langle \mathbf{hx}_j \rangle, \langle \mathbf{ty}_j \rangle, y_j)$ atom.
- *How to "invent" or re-use appropriate predicates? How to implement "invented" predicates?* This so-called predicate invention problem is tackled at four points during the synthesis. At Steps 2 and 3, the Database Method re-uses predefined predicates: this amounts to the use of domain knowledge. At Step 6, the Synthesis Method (see Section 14.2.4 below) invents new composition operators. As of now, there is however no method of re-using common composition operators. At Step 7, the Proof-as-Programs Method synthesizes discriminants that are in terms of predicates other than *r*/*n*.
- *How to discover which parameters are auxiliary parameters?* Due to our restriction to version 3 of the divide-and-conquer schema, auxiliary parameters are not taken into account. See Section 14.2.2 below on how to achieve this.
- *How to synthesize logic algorithms that are non-deterministic?* Task G of Step 4 is the first—and only—one to discover whether the intended relation is non-deterministic given a ground value of the induction parameter and variables for the other parameters, namely because the *exp*/3 operator discovers sets of potential values of the ***TY*** in some non-minimal disjuncts of $exp(LA_4(r))$. But, from Table 14-2, we observe that $LA_4(r)$ is at best deterministic, and that it is Step 5 that produces the first possibly non-deterministic logic algorithm. Indeed, given a ground value of the induction parameter, a logic algorithm can only be non-deterministic if it actually unifies *Y* with some values in several non-mutually-exclusive cases. This is clearly not the case with $LA_4(r)$, where only the induction parameter *X* is unified with some values. But such is the case with $LA_5(r)$,

where the instances of the $SolveNonMin_k$ necessarily do unify Y with some values, and this possibly non-deterministically (because of the MSG Method). Note that the disjunction between the minimal case and the non-recursive case is not an additional source of non-determinism of $LA_5(r)$, because these cases are known to be mutually exclusive (because of Step 2). Also note that the non-determinism due to the instances of the $SolveNonMin_k$ might even disappear at Step 7 (Synthesis of the $Discriminate_k$), namely if the added discriminants render these sub-cases mutually exclusive. Step 6 acts in exactly the same way as Step 5: the instances of the $ProcComp_k$ necessarily do unify Y with some values, and this possibly non-deterministically (because of the MSG Method). And the non-determinism due to the instances of the $ProcComp_k$ might also disappear at Step 7, namely if the added discriminants render these sub-cases mutually exclusive. All this shows that the discovery of non-determinism and its "synthesis" take place at different moments. They are even totally unrelated events.

- *How to achieve a synthesis that yields logic algorithms that are correct wrt their specifications?* This is impossible to guarantee with specifications by examples and properties. However, the stepwise synthesis framework of Section 7.3.2 and the correctness theorems of the seven synthesis steps clearly identify the critical points, where interaction with the specifier should thus take place.

We can also analyze how some of the desired features enumerated in Section 11.1 have been achieved (note that most of them were decisions rather than open questions):

- The *degree of automation* depends on the actual implementation of the synthesis mechanism. Indeed, as it stands, the mechanism is fully automatable. However, there are critical points where interaction with the specifier is safer than blind application of the listed heuristics, however reliable they may seem. The synthesis mechanism is example-driven, and uses the properties at selected moments. This implies that synthesis cannot be done from properties alone. On the other hand, if only examples are provided, or if "not enough" properties are provided, then there should be a lot of interaction with the specifier.
- The *kinds of inference* used during the synthesis are inductive inference from the examples (due to the usage of the MSG Method), and deductive inference from the properties (due to the usage of the Proofs-as-Programs Method). The synthesis mechanism is a hybrid of transformational synthesis (due to its stepwise approach), proofs-as-programs synthesis (due to the way discriminants are synthesized), knowledge-based synthesis (due to the usage of the divide-and-conquer schema and typed data-bases), bottom-up approximation-driven empirical learning (due to its usage of incomplete properties), and Summers' recurrence detection mechanism. But it is not really comparable to the other approaches surveyed in Chapter 3, namely Biermann's function merging mechanism (due to the presence of multiple examples), and Shapiro's debugging mechanism (due to the non-incremental approach, and to the absence of counter-examples). The mechanism seems very robust to example ordering (due to its non-incrementality) and example choice.

Note that the expansion phase of synthesis may be likened to *trace generation*, while the reduction phase of synthesis may be likened to *trace generalization* (see Section 3.3).

14.2 Extensions

Let's now discuss some extensions to the existing synthesis mechanism. First, in Section 14.2.1, we present the problems posed by the introduction of negation in examples, properties, and logic algorithms. In Section 14.2.2, we show how the mechanism can be en-

hanced to better cope with auxiliary parameters. We then see how the preliminary restrictions of Section 11.2 can be overcome:

- the support of schemas other than version 3 of the divide-and-conquer schema is discussed in Section 14.2.3;
- the synthesis of multiple-loop logic algorithms is discussed in Section 14.2.4, where we describe the Synthesis Method as a complement to the MSG Method;
- the support of inductively defined data-types other than integers and lists is extremely modular: just extend the typed databases with the appropriate instances.

Finally, in Section 14.2.5, we generalize the computational contents of properties, and examine what would need to be done to support such properties. The Future Work sections of the Building Blocks chapters of Part II of course constitute other directions for future research.

14.2.1 Negation

In Section 11.1, we have constrained the bodies of logic algorithms and properties to be free of negation, namely because the Proofs-as-Programs Method (which is used by Step 7 of the synthesis mechanism) cannot handle negation. As outlined in Section 9.5, this method could be extended to handle negated atoms that have primitive predicates, which thus overcomes the overall restriction.

Note however that negation is theoretically useless: Horn clause logic is a universal computing formalism (see [Hogger 84] or [Lloyd 87] for accounts of this result). Negation thus doesn't increase the expressive power of Horn clause logic. In practice, however, statements and algorithms are often significantly simplified by the use of negation.

Negative examples (also called *counter-examples*) have been omitted from the specification format adopted in Section 11.1. We are now able to provide the motivation for this decision. Counter-examples are in fact useless in a divide-and-conquer approach, because the relation is "built" from positive examples. And only positive examples can possibly be used in order to "explain" how a specific positive example is "built". Counter- examples could however be used to check whether the synthesized logic algorithm isn't too general compared to the intended relation. This is subject to future research.

Note that if properties could also be equivalence statements rather than only Horn clauses, then properties would actually be a generalization of negative examples, because they would then embody conditions about when the intended relation does not hold. Most properties are actually equivalence statements. For instance, all properties in the sample specifications of Section 6.2, except for the "informative" property P_2 of *EP(efface)*, would also be correct if they were equivalence statements. The current version of the synthesis mechanism would only use the *if* parts of such extended properties in Step 7 for the synthesis of the discriminants. But an extended version performing the above-mentioned over-generalization-check could use the *only-if* parts to infer negative examples, rather than relying on their explicit presence in the specification.

14.2.2 Auxiliary Parameters

Throughout this thesis, the phrase "auxiliary parameter" keeps popping up. Intuitively, an *auxiliary parameter* is a parameter that has nothing to do with the inductive nature of the relation. Note that a parameter is auxiliary for a relation, and hence for all possible logic algorithms for that relation. Logic algorithm synthesis by induction on an auxiliary parameter is obviously a bad idea.

So far, we have completely ignored auxiliary parameters. This is justifiable by the observation that their identification is not necessary at all for correct algorithm design. Indeed, as the logic algorithms of Section 5.2 show, it is possible to write algorithms that don't distin-

guish between auxiliary parameters and "ordinary" parameters. However, the (de)composition of an auxiliary parameter from (into) its heads and tails may look cumbersome because an auxiliary parameter *Y* and its tail *TY* are eventually found (by Step 6) to be identical: *Y=TY*. But it is precisely this composition pattern that allows the detection of auxiliary parameters, and their elimination from the decomposition machinery, thus transforming the logic algorithm into a more "graceful" and "natural" version. We call this *late detection*, because it only allows the transformation of the synthesized logic algorithm, rather than a simplification of its actual synthesis process. The appearance of *Y=TY* in all recursive cases of a logic algorithm is assumed to be the formal definition of the notion of auxiliary parameter.

We should however not forget that just a casual glance at a specification will not always tell whether a parameter is an "ordinary" or an auxiliary one. Things are even more difficult with automated algorithm design, and the surest way is indeed to ignore the potential existence of auxiliary parameters until a transformation phase. But suppose now that knowledge about which parameters are auxiliary parameters is available earlier during the algorithm synthesis process. It would certainly be helpful to pre-compile the then needed transformations into a schema with an explicit consideration of auxiliary parameters. The benefit would be less complex general example sets for the synthesis of the $Process_k$ and $Compose_k$, and hence smaller search spaces. The modified version 4 of the divide-and-conquer schema is:

```
R(X,Y,Z) ⇔
            Minimal(X)        ∧ Solve(X,Y,Z)
   ∨ ∨1≤k≤c NonMinimal(X)     ∧ Decompose(X,HX,TX)
                              ∧ Discriminatek(HX,TX,Z)
                              ∧ (      SolveNonMink(HX,TX,Y,Z)
                                   |
                                      R(TX,TY,Z)
                                   ∧ Processk(HX,HY)
                                   ∧ Composek(HY,TY,Y,Z)     )
```

where ***Z*** is the vector of auxiliary parameters. Note how ***Y*** has disappeared from the discriminants: this knowledge is encoded as Heuristic 13-4 at Step 7 (Synthesis of the $Discriminate_k$). This revised schema also calls for a revised admissibility criterion.

Now, how could knowledge about which parameters are auxiliary parameters be available earlier during the algorithm synthesis? There are basically two solutions:

- *declaration*: the specifier could declare them as such at specification time;
- *early detection*: at Step 4 (Syntactic introduction of the recursive atoms), a parameter that is not of an inductive type must definitely be an auxiliary parameter; this heuristic is sound, but not complete. Other (possibly interactive) heuristics could be elaborated.

These solutions allow faster synthesis and more "natural" synthesized logic algorithms.

14.2.3 Supporting Other Schemata

The presented synthesis mechanism is guided by version 3 of the divide-and-conquer logic algorithm schema. This preliminary restriction(made in Section 11.2) has considerably simplified the notations needed for the theoretical presentation. The support of version 4 (relations of any non-zero arity) is actually a pretty straightforward extension, because only some additional vectorization is needed. Version 4 is actually supported by the implementation of the synthesis mechanism.

The support of version 5 (any number of minimal or non-minimal forms) and version 6 (compound induction parameters) is considered future research, as considerable extensions to the already defined tasks need to be developed. Note however that Section 5.2.3 shows that

single minimal forms and non-minimal forms are more general than one might believe at first sight. As outlined in Section 8.4, there is still a lot of space for designing even more sophisticated divide-and-conquer schemas.

However, version 3 of the divide-and-conquer schema is hardwired into our synthesis mechanism. This results from a hardwired sequence of instantiations of predicate variable of that schema, as well as from a hardwired mapping between these predicate variables and the methods of the developed tool-box. Parameterizing the synthesis mechanism on algorithm schemas and tool-boxes would thus be a first step towards supporting schemas reflecting algorithm design strategies other than divide-and-conquer.

In other words, a Step 0 would be to select an appropriate schema, and the subsequent steps would be either a hardwired sequence (specific to the selected schema) of applications of methods, or a user-guided selection of variables and methods. Our grand view of algorithm synthesis systems thus is one of a large workbench with a disparate set of highly specialized methods for a set of schemas that covers (as much as possible of) the space of all possible algorithms.

In defense of our hardwiring the divide-and-conquer schema, we should however make the following two remarks. First, the hardwired sequence of predicate variable instantiations is justifiable by our arguing (see Section 11.3 and Section 12.3.2) that this sequence is probably the only one in the context of logic programming. Second, the hardwired mapping between the predicate variables and the methods of the tool-box is justifiable by pure common sense.

14.2.4 The Synthesis Method

Within a restricted setting, the MSG Method (see Chapter 10) infers, from a finite set of general examples, a non-recursive logic algorithm that is defined in terms of the =/2 primitive only, and that is correct wrt a natural extension of the given examples. But if the resulting logic algorithm is judged, by some application-dependent heuristic (such as Heuristic 13-2), to be "not good enough", then the assumption that the examples can be covered by such a logic algorithm must be revised: recursion, or other predicates, or both, might be needed. This requires a more involved method.

A possible solution is to automatically infer a property set of the intended relation and then to use an entire synthesis mechanism (such as the one described in the two previous chapters) in order to infer a logic algorithm that is complete wrt the given examples and inferred properties. We here describe such a method, called the *Synthesis Method*, which is specific to the synthesis of the $Process_k$ and $Compose_k$ predicate variables of the divide-and-conquer schema.

The Synthesis Method assumes that there is a single sub-case of the recursive case (w=1), and that $procComp_1(\boldsymbol{HX},\boldsymbol{TY},Y)$, which is hereafter denoted $procComp(\boldsymbol{HX},\boldsymbol{TY},Y)$, is defined by a full-fledged recursive logic algorithm. The synthesis of *LA(procComp)* may thus be achieved by the following sequence of tasks:

- Task S': Inference of $\mathcal{E}(procComp)$ from $LA_5(r)$ and $EP(r)$;
- Task T': Inference of $\mathcal{P}(procComp)$ from $LA_5(r)$ and $EP(r)$;
- Task U': Synthesis of *LA(procComp)* from *EP(procComp)*.

Let's explain these tasks one by one, then illustrate them on two sample problems, and finally assess this new method.

Task S': Inference of an Example Set

The inference of an example set would be identical to the method of Task S of Step 6 (Synthesis of the $Process_k$ and $Compose_k$) if the synthesis mechanism were able to handle general

examples. Such is however not the case. But we can exploit the fact that the Synthesis Method is only invoked upon "dislike" of the result produced by the MSG Method. Indeed, remember that the non-ground case of the MSG Method proceeds by invoking its other case for every atom set of *adm(G(procComp))*, and then selecting a "best" solution. So if the logic algorithm resulting from the MSG Method is judged to be "not good enough", then it would be most reasonable to give the corresponding atom set a "second chance" (provided it is an example set), rather than to take any other atom set that corresponds necessarily to a "less good" solution of the MSG Method. Let *E(procComp)* be the chosen set.

Task T': Inference of a Property Set

The inference of a property set is based on the observation that the properties of the original predicate are "inherited" by the properties of the *procComp*/3 predicate. This can be formalized as follows.

For every property:

$$r(s_i,t_i) \Leftarrow B_i \tag{1}$$

such that:

$$nonMinimal(s_i) \tag{2}$$

$$decompose(s_i,\mathbf{hs}_i,\mathbf{ts}_i) \tag{3}$$

where *nonMinimal*/1 is the instance of *NonMinimal*/1 selected at Step 2, and *decompose* is the deterministic instance of *Decompose* selected at Step 3,
find variants of examples or body-less properties:

$$r(ts_{ij},tt_{ij}) \tag{4}$$

and infer the following property of *procComp*:

$$procComp(\mathbf{hs}_i,\mathbf{tt}_i,t_i) \Leftarrow B_i \tag{5}$$

This can be justified as follows. By unfolding the head of (1) according to the necessarily unique recursive disjunct of $LA_6(r)$, as it will be, we obtain:

$$nonMinimal(s_i) \wedge decompose(s_i,\mathbf{HS},\mathbf{TS}) \wedge \mathbf{r(TS,TT)} \wedge procComp(\mathbf{HS},\mathbf{TT},t_i) \Leftarrow B_i$$

By (2), (3), and the determinism of *decompose*, this simplifies into:

$$\mathbf{r(ts}_i\mathbf{,TT)} \wedge procComp(\mathbf{hs}_i,\mathbf{tt}_i,t_i) \Leftarrow B_i$$

By (4), this effectively reduces to (5).

Moreover, the resulting properties can often be generalized, for instance by replacing base-case constants by variables. Such a generalization process should be interactive, and should yield new properties that are consistent with the examples.

Task U': Synthesis of LA(procComp)

A logic algorithm *LA(procComp)* can now be synthesized from the inferred specification by examples and properties. We use the synthesis mechanism described in the two previous chapters for doing so. The predicate *procComp* is assumed to be a new primitive.

This completes Step 6, and the synthesis of *LA(r)* proceeds to Step 7 (Synthesis of the $Discriminate_k$), where a *true* discriminant is added because the information of the properties of *r*/*n* is "inherited" by the properties of *procComp*, and thus already accounted for.

Illustration

Let's illustrate the described method on two sample problems.

Example 14-1: The synthesis of *LA(permutation-int-L)* leads to the rejection of the first *LA(pcPermutation)* that is synthesized by the MSG Method. This is shown in Example 13-4. The Synthesis Method is thus invoked. The specification *EP(permutation)* is as in Example 6-8. The logic algorithm LA_6*(permutation)* will be:

```
permutation(L,P) ⇔
    L=[]       ∧ P=[]                                  {E1}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ permutation(TL,TP)
               ∧ pcPermutation(HL,TP,P)          {E2-E10}
```

At Task S', the inferred example set is the one corresponding to that first solution of the MSG Method, namely:

```
ℰ(pcPermutation)={ pcPermutation(a,[],[a])                (F2)
                   pcPermutation(b,[c],[b,c])             (F3)
                   pcPermutation(b,[c],[c,b])             (F4)
                   pcPermutation(d,[e,f],[d,e,f])         (F5)
                   pcPermutation(d,[f,e],[d,f,e])         (F6)
                   pcPermutation(d,[e,f],[e,d,f])         (F7)
                   pcPermutation(d,[e,f],[e,f,d])         (F8)
                   pcPermutation(d,[f,e],[f,d,e])         (F9)
                   pcPermutation(d,[f,e],[f,e,d]) }      (F10)
```

where example F_i corresponds to example E_i. Note that F_5 and F_6 are "variants". The same holds for F_7 and F_9, and for F_8 and F_{10}. But we cannot eliminate "variants" because we cannot know that *pcPermutation*/3 effectively is a structural-manipulation problem.

At Task T', the inference of a property set goes as follows:

- from property P_1 of $\mathcal{P}$*(permutation)*:

  permutation([*X*],[*X*])

  infer property Q_1 of $\mathcal{P}$*(pcPermutation)*:

  pcPermutation(*X*,[],[*X*])

  because example E_1 of $\mathcal{E}$*(permutation)* shows that a permutation of [] is [];
- similarly, from property P_2 of $\mathcal{P}$*(permutation)*:

  permutation([*X*,*Y*],[*X*,*Y*])

  infer property Q_2 of $\mathcal{P}$*(pcPermutation)*:

  pcPermutation(*X*,[*Y*],[*X*,*Y*])

  because property P_1 of $\mathcal{P}$*(permutation)* shows that a permutation of [*Y*] is [*Y*];
- finally, from property P_3 of $\mathcal{P}$*(permutation)*:

  permutation([*X*,*Y*],[*Y*,*X*])

  infer property Q_3 of $\mathcal{P}$*(pcPermutation)*:

  pcPermutation(*X*,[*Y*],[*Y*,*X*])

  because property P_1 of $\mathcal{P}$*(permutation)* shows that a permutation of [*Y*] is [*Y*].

All three properties could be generalized by replacing all occurrences of the constant *nil* by the same new variable, but this isn't even necessary here.

At Task U', the entire synthesis mechanism of the two previous chapters is used to synthesize *LA(pcPermutation)* from *EP(pcPermutation)* as generated by the two previous tasks. Supposing an induction on the second parameter and an intrinsic decomposition thereof, plus a detection that the first parameter necessarily is an auxiliary parameter, the result is:

```
pcPermutation(E,L,R) ⇔
    L=[]       ∧ E=_ ∧ R=[E]                              {F2}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ true
               ∧ E=_ ∧ R=[E|L]                        {F3,F5,F6}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ true
               ∧ pcPermutation(TE,TL,TR)
               ∧ E=TE ∧ R=[HL|TR]          {F4,F7,F8,F9,F10}
```

Note that *EP(pcPermutation)* and *LA(pcPermutation)* indicate that the underlying problem is the one described in the informal specification of *stuff*/3.

The synthesis of *LA(permutation)* proceeds to Step 7 (Synthesis of the *Discriminate*$_k$), where *LA*$_7$*(permutation)* is found to be equivalent to *LA*$_6$*(permutation)*. This resulting logic algorithm for *permutation*/2 is correct, but yields redundant solutions if some element has multiple occurrences in *L*. The usage of *efface(HL,P,TP)*, as in *LA(permutation-L)* (LA 5-14), which is stronger than *stuff(HL,TP,P)*, remedies to such redundancy. This version may be obtained by our synthesis mechanism if the condition $X \neq Y$ is added to property P_3 of *EP(permutation)*. ♦

Example 14-2: The synthesis of *LA(compress-int-C)* leads to the rejection of *LA(pcCompress)* as synthesized by the MSG Method. Indeed, the msgs are:

```
pcCompress(A,1,T,[A|T])
pcCompress(A,2,T,[A,A|T])
pcCompress(e,3,T,[e,e,e|T])
```

There are thus as many cliques as properties, and Heuristic 13-2 applies. The Synthesis Method is thus invoked. The specification *EP(compress)* is as in Example 6-1. The logic algorithm *LA*$_6$*(compress)* will be:

```
compress(L,C) ⇔
    C=[]       ∧ L=[]                                     {E1}
  ∨ C=[_,_|_]  ∧ C=[HC1,HC2|TC]
               ∧ compress(TL,TC)
               ∧ pcCompress(HC1,HC2,TL,L)             {E2-E8}
```

where HC_1 is a value and HC_2 is a counter.

At Task S', the inferred example set is the one corresponding to the solution of the MSG Method, namely:

```
E(pcCompress) = { pcCompress(a,1,[],[a])                  (F2)
                  pcCompress(b,2,[],[b,b])                (F3)
                  pcCompress(c,1,[d],[c,d])               (F4)
                  pcCompress(e,3,[],[e,e,e])              (F5)
                  pcCompress(f,2,[g],[f,f,g])             (F6)
                  pcCompress(h,1,[i,i],[h,i,i])           (F7)
                  pcCompress(j,1,[k,m],[j,k,m]) }         (F8)
```

where example F_i corresponds to example E_i.

At Task T', the inference of a property set yields:

```
P(pcCompress) = { pcCompress(X,1,[],[X])                  (Q1)
                  pcCompress(X,2,[],[X,Y]) ⇐ X=Y          (Q2)
                  pcCompress(X,1,[Y],[X,Y]) ⇐ X≠Y }       (Q3)
```

where property Q_i results from property P_i. All three properties could be generalized by replacing all occurrences of the constant *nil* by a new variable, but this isn't even necessary.

At Task U', the entire synthesis mechanism of the two previous chapters is used to synthesize *LA(pcCompress)* from *EP(pcCompress)* as generated by the two previous tasks. Supposing an induction on the second parameter, and an intrinsic decomposition, the result is:

```
pcCompress(E,N,L,R) ⇔
    N=s(0)     ∧ E=_ ∧ L=_ ∧ R=[E|L]
               ∧ (L=[]) ∨ (L=[HL|_] ∧ HL≠E)  {F2,F4,F7,F8}
  ∨ N=s(s(_)) ∧ N=s(TN) ∧ HN=N
               ∧ pcCompress(TE,TN,TL,TR)
               ∧ E=TE ∧ L=TL
                    ∧ R=[E|TR] ∧ TR=[E|_]      {F3,F5,F6}
```

Note that *V* and *L* are auxiliary parameters. The synthesis of *LA(compress)* proceeds to Step 7 (Synthesis of the $Discriminate_k$), where $LA_7(compress)$ is found to be equivalent to $LA_6(compress)$. ♦

Assessment

The Synthesis Method allows us to overcome the preliminary restriction of Section 11.2 to the synthesis of single-loop logic algorithms only. Indeed, if the instance of *ProcComp* is defined by a full-fledged recursive logic algorithm, then *LA(r)* is a multiple-loop logic algorithm. The synthesis mechanism is thus recursively defined, and there should be no limit to the number of nested loops it could support.

But why not immediately use the Synthesis Method? The reason is that both methods tackle different classes of problems. The MSG Method and the Synthesis Method are a joint answer to the same problem: how to infer, from a finite set of general examples of an unknown relation that is however known to feature a given construction pattern between its parameters, a logic algorithm that is correct wrt a natural extension of the given examples. The MSG Method is the "base case" of the answer, because it doesn't look for recursion, and the Synthesis Method is the "structure case" of the answer, because it does look for recursion.

Note that the instances of *ProcComp* that are synthesized by the Synthesis Method are often one of the following:

- partially evaluated versions of logic algorithms that a human algorithm designer would have (re-)used; for instance, a version of *LA(append)* where the first parameter is always a non-empty list;
- (loop-)mergers of several logic algorithms that a human algorithm designer would have (re-)used; for instance, *LA(pcCompress)* above is the loop-merger of the conjunction used in *LA(compress-int-C)* (LA 5-2):

  $$plateau(N,E,HL) \wedge firstPlateau(L,HL,TL)$$

  which a human algorithm designer would probably even have generalized to:

  $$plateau(N,E,HL) \wedge append(HL,TL,L)$$

  by relaxing, if not "forgetting", the fact that *HL* must be a plateau;
- a combination of the above.

This clearly shows the impeccable precision of automated synthesis compared to manual construction.

In terms of related research, other synthesis systems (such as CYPRESS [Smith 85]) are recursively defined, and hence infer their own specifications for appearing sub-problems.

The Synthesis Method is currently too closely entangled with the divide-and-conquer schema, with the synthesis mechanism described in the two previous chapters, and with the objective of Step 6 of that mechanism. In order to make it a full-fledged stand-alone method that can be added to our general tool-box, some future research is needed to make it schema-independent. It will of course still rely on the MSG Method for the inference of an example set. But the inference of a property set could probably be made predicate-variable-independent and schema-independent by some theorem proving task that exploits the integrity constraints of a schema.

Moreover, even the <MSG Method, Synthesis Method> couple isn't the complete answer to the synthesis of instances of the $ProcComp_k$ predicate variables. Indeed, some instances might be defined neither in terms of =/2 only (and hence the MSG Method is inappropriate), nor by recursion (and hence the Synthesis Method is inappropriate).

Example 14-3: A good illustration of this phenomenon is the *minimum(L,M)* relation, which may be defined by the following divide-and-conquer logic algorithm:

```
minimum(L,M) ⇔
    L=[_]       ∧ L=[E] ∧ M=E
  ∨ L=[_,_|_]   ∧ L=[HL|TL]
                ∧ minimum(TL,TM)
                ∧ pcMinimum(HL,TM,M)
```

where *pcMinimum*/3 is specified as follows:

pcMinimum(A,B,M) iff *M* is the minimum of *A* and *B*,
where *A*, *B*, and *M* are integers.

which is defined by the following logic algorithm:

```
pcMinimum(A,B,M) ⇔
    A≤B ∧ M=A
  ∨ A>B ∧ M=B
```

which is clearly not producible by either the MSG Method, or the Synthesis Method. ♦

This shows the need for even more methods to complement the existing ones. Good candidates would be a *Reuse Method* (looking for instances of atoms listed in a database of typical composition predicates), and an *Analogy Method* (looking for analogies between the examples in order to infer instances of the $ProcComp_k$). In the absence of "good" answers, the last straw is to ask the specifier.

14.2.5 The Computational Contents of Properties

It is very important to understand that the MSG Method (as an inductive technique that is based on examples) and the Proofs-as-Programs Method (as a deductive technique that is based on properties) are not at all tied to Steps 5+6 and 7, respectively. As general methods of a tool-box, they are actually often interchangeably applicable for the instantiation of any predicate variable (of any schema).

For instance, the Proofs-as-Programs Method could be invoked right after Step 4 (Syntactic introduction of the recursive atoms) in order to instantiate *Solve*, the $SolveNonMin_k$, the $ProcComp_k$, and the $Discriminate_k$ from $\mathcal{P}$*(firstPlateau)* and LA_4*(firstPlateau)*, and thus do the work of Steps 5 and 6 as well.

Of course, appropriate generalization heuristics would have to be devised. But the point is that this is sometimes possible, and sometimes even desirable.

The fundamental observation underlying such "cross-uses" of methods is the following. Given a problem and the decision to perform a divide-and-conquer design, there are different

kinds of information to be conveyed in the specification, namely *solving*, *discrimination*, *processing*, and *composition* information. These kinds of information are such-named because their contents wind up in the instantiations of the similar-named predicate variables. So, with specifications by examples and properties, the specifier has many alternatives about which kind of information to make explicit in the bodies of properties. These kinds of information are (often) independent of the selected induction parameter and the selected decomposition strategy.

For instance, in *EP(firstPlateau)* as listed in Example 6-2, property P_1 carries solving information, while P_2 and P_3 carry discrimination information in their bodies: these particular contents justify the particular choice of methods at Steps 5 to 7.

The so far implicit assumption behind the presented synthesis mechanism is that properties only carry solving information, or discrimination information, or both. But given the interchangeability of methods and this variability of specifications, it is obvious that the results presented here are highly adaptable to different assumptions. However, in order to support properties with processing or composition information, the generalization heuristics need to be adapted. Moreover, ways of detecting the presence of such extended computational contents of properties need to be devised.

Handling Properties with Processing Information

The following example illustrates the need for properties with processing information.

Example 14-4: Consider the following informal specification:

triples(L,R) iff *R* is the elements of *L* multiplied by 3,
where *L* and *R* are integer lists.

A sample specification by examples and properties is:

```
ℰ(triples) = {  triples([],[])                          (E1)
                triples([1],[3])                        (E2)
                triples([3,2],[9,6])                    (E3)
                triples([5,3,6],[15,9,18]) }            (E4)

𝒫(triples) = {  triples([X],[Y]) ⇐ mult(X,3,Y) }        (P1)
```

where *mult(A,B,P)* holds iff *P* is the product of the integers *A* and *B*. Property P_1 obviously carries processing information.

Assuming a synthesis by intrinsic induction on *L*, at Step 6 (Synthesis of the $Process_k$ and $Compose_k$), the targeted logic algorithm is:

```
triples(L,R) ⇔
    L=[]       ∧ R=[]                                  {E1}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ triples(TL,TR)
               ∧ pcTriples(HL,TR,R)                  {E2-E4}
```

where *LA(pcTriples)* is yet to be synthesized. The informal specification is that *R* is *TR* preceded by *HL* multiplied by 3. The corresponding examples are:

```
ℰ(pcTriples)= { pcTriples(1,[],[3])                     (F2)
                pcTriples(3,[6],[9,6])                  (F3)
                pcTriples(5,[9,18],[15,9,18]) }         (F4)
```

The MSG Method finds a single clique, whose msg is *pcTriples(A,T,[B|T])*. So LA_6*(triples)* is defined as follows (after some rewriting):

```
triples(L,R) ⇔
    L=[]      ∧ R=[]                                  {E1}
  ∨ L=[_|_]   ∧ L=[HL|TL]
              ∧ triples(TL,TR)
              ∧ R=[HR|TR]                             {E2-E4}
```

Note that *HR* is not yet linked to the computations acting on *L*. This holds because the instantiation of *pcTriples*/3 found by the MSG Method is in fact too general: *R*=[*HR*|*TR*] correctly performs the composition part, but also incorrectly assumes that *HR*=_ does the job for the processing part.

The synthesis proceeds to Step 7 (Synthesis of the $Discriminate_k$). Property P_1 is found to be a logical consequence of clause C_2, which is generated from the second disjunct of LA_6*(triples)*:

```
triples([X],[Y]) ← mult(X,3,Y)
      DCI: C2    ↓    {}
[X]=[?_|?_] & [X]=[?HL|?TL] & triples(?TL,?TR) &
      [Y]=[?HR|?TR]  ←  mult(X,3,Y)
3 × DCI: LA(=)   ↓    {HL/X, TL/[], TR/[], HR/Y}
triples([],[])  ←  mult(X,3,Y)
      DCI: E1    ↓    {}
         ←  mult(X,3,Y)
```

The extracted clause is:

```
discTriples2(L,R,HL,TL,TR,HR) ←
                    L=[X],R=[Y],
                    HL=X,TL=[],TR=[],HR=Y,
                    mult(X,3,Y)
```

However, the extracted clause defines $Process_2$ rather than $Discriminate_2$. The generalization heuristics of Section 13.3.2 should thus not be used here, because they are tailored for the generalization of discriminating clauses. An adaptation of these heuristics to the generalization of processing clauses is straightforward, and yields here:

```
procTriples2(HL,HR) ← mult(HL,3,HR)
```

Hence LA_7*(triples)* is defined as follows (after some rewriting):

```
triples(L,R) ⇔
    L=[]      ∧ R=[]                                  {E1}
  ∨ L=[_|_]   ∧ L=[HL|TL]
              ∧ triples(TL,TR)
              ∧ mult(HL,3,HR)
              ∧ R=[HR|TR]                             {E2-E4}
```

This brings *HR* into the flow of computations acting on *L*, and thus corrects the instantiation of *pcTriples*/3. ♦

Handling Properties with Composition Information

On the other hand, one might argue that properties with composition information would prejudice over choices that should rather be made by the synthesis mechanism, such as the selection of an induction parameter and of its decomposition strategy. This is however not always the case, as illustrated by the following two sample syntheses.

Example 14-5: Consider the following property:

```
permutation([X,Y],R) ⇐ stuff(X,[Y],R)
```

Partial evaluation of *LA(stuff)* and unfolding of the *stuff*/3 atom transform the property into the following statement:

```
permutation([X,Y],R) ⇐ R=[X,Y] ∨ R=[Y,X]
```

This statement is no longer a property (because of the disjunction in the body), but it is equivalent to the conjunction of properties P_2 and P_3 of *EP(permutation)*, as in Example 6-8. There is thus no extra information in the above property, and it doesn't force at all the selection of the first parameter as induction parameter, nor its intrinsic decomposition. ♦

Example 14-6: Consider the following specification:

```
ℰ(sum) = {  sum([],0)                          (E1)
            sum([7],7)                         (E2)
            sum([4,5],9)                       (E3)
            sum([2,3,1],6) }                   (E4)
𝒫(sum) = {  sum([X],X)                         (P1)
            sum([X,Y],R) ⇐ add(X,Y,R) }        (P2)
```

where *sum(A,B,S)* holds iff *S* is the sum of the integers *A* and *B*. Property P_2 obviously carries composition information. There is no possible partial evaluation of P_2.

Suppose a synthesis with induction on *L* and intrinsic decomposition. We now show the usefulness of P_2.

First, the omission of P_2 would, at Step 4 (Syntactic introduction of the recursive atoms), lead to the need for specifier intervention, because the deductive oracle (based on *EP(sum)*) cannot compute *TS* for example E_4. Indeed, no example or property can infer that *TS*=4 for *TL*=[3,1], such that *sum(TL,TS)* holds.

Second, supposing P_2 <u>is</u> present, then at Step 6 (Synthesis of the $Process_k$ and $Compose_k$) the targeted logic algorithm LA_6*(sum)* is:

```
sum(L,S) ⇔
    L=[]        ∧ S=[]                          {E1}
  ∨ L=[_|_]     ∧ L=[HL|TL]
                ∧ sum(TL,TS)
                ∧ pcSum(HL,TS,S)                {E2-E4}
```

where *LA(pcSum)* is yet to be synthesized. The intended instantiation for *pcSum*/3 obviously is *add*/3. The corresponding examples are:

```
ℰ(pcSum)= { pcSum(7,0,7)                        (F2)
            pcSum(4,5,9)                        (F3)
            pcSum(2,4,6) }                      (F4)
```

The MSG Method finds 3 cliques, each containing one example. Heuristic 13-2 suggests rejecting this result, and invoking the Synthesis Method. The following properties are inferred:

```
𝒫(pcSum)= { pcSum(X,0,X)                        (Q1)
            pcSum(X,Y,R) ⇐ add(X,Y,R) }         (Q2)
```

The synthesis of *LA(pcSum)* by intrinsic induction on the second parameter yields at Step 6:

```
pcSum(A,B,R) ⇔
    B=0         ∧ R=A                           {F2}
  ∨ B=s(_)      ∧ B=s(TB) ∧ B=HB
                ∧ pcSum(TA,TB,TR)
                ∧ pcPcSum(HB,TA,TR,A,R)         {F3-F4}
```

where *LA(pcPcSum)* is yet to be synthesized. The corresponding examples are actually constrained general examples (whose handling is also considered future work):

$$\mathcal{C}(\texttt{pcPcSum}) = \{\ \texttt{pcPcSum(5,TA}_1\texttt{,TR}_1\texttt{,4,9) [add(TA}_1\texttt{,4,TR}_1\texttt{)]}\ (C_3)$$
$$\texttt{pcPcSum(4,TA}_2\texttt{,TR}_2\texttt{,2,6) [add(TA}_2\texttt{,3,TR}_2\texttt{)]}\}\ (C_4)$$

These examples have admissible alternatives under the following type constraints:

$$0 \le TA_1 \le 4 \quad \text{and} \quad 0 \le TR_1 \le 9$$
$$0 \le TA_2 \le 2 \quad \text{and} \quad 0 \le TR_2 \le 6$$

Under all these constraints, C_3 has 5 admissible alternatives, while C_4 has 3 admissible alternatives. Among the $5 \times 3 = 15$ admissible alternatives of $\mathcal{C}$*(pcPcSum)*, the MSG Method locates 1 that has 1 clique (namely when TA_1=4, TR_1=8, TA_2=2, and TR_2=5), whereas the other 14 have 2 cliques. Selecting the solution with the least number of cliques, the MSG Method infers the following logic algorithm for *pcPcSum*/5:

```
pcPcSum(HB,TA,TR,A,R) ⇔
                HB=_ ∧ A=TA ∧ R=s(TR) ∧ TA≥2 ∧ TR≥5
```

where the last two atoms should obviously be generalized away. Hence LA_6*(pcSum)* looks as follows (note that *A* turns out to be an auxiliary parameter):

```
pcSum(A,B,R) ⇔
    B=0        ∧ R=A                                   {F2}
  ∨ B=s(_)     ∧ B=s(TB) ∧ B=HB
               ∧ pcSum(TA,TB,TR)
               ∧ A=TA ∧ R=s(TR)                      {F3-F4}
```

The synthesis of *LA(pcSum)* proceeds to Step 7 (Synthesis of the $Discriminate_k$). Property Q_1 is a logical consequence of the first disjunct of LA_6*(pcSum)*, but the extracted discriminant is redundant with the already existing atoms in that disjunct. Property Q_2 may be proven by an inductive proof (which is also considered future work) to be a logical consequence of LA_6*(pcSum)*. Again, the extracted discriminant is redundant with the already existing atoms. Hence LA_7*(pcSum)* is equivalent to LA_6*(pcSum)*.

Note that if property P_2 were expressed as an equivalence statement, then property Q_2 would also be an equivalence statement, and no synthesis would be needed. But our synthesis mechanism doesn't exploit such opportunities: it rather prefers to re-invent the wheel and then verify whether it is a wheel.

This completes the synthesis of *LA(pcSum)*. The synthesis of *LA(sum)* proceeds to Step 7 (Synthesis of the $Discriminate_k$). Both properties P_1 and P_2 are found to be logical consequences of LA_6*(sum)*, and the extracted discriminants are redundant with the already existing atoms. Hence LA_7*(sum)* is equivalent to LA_6*(sum)*.

This completes the synthesis of *LA(sum)*. However, the re-discovery of *add*/3 for *pcSum*/3 is quite tedious. It would have been more elegant to immediately invoke the Proofs-as-Programs Method at Step 6 (Synthesis of the $Process_k$ and $Compose_k$), instead of the MSG Method first. This would go as follows.

Property P_1 is a logical consequence of clause C_2, which is generated from the second disjunct of LA_6*(sum)*, but the extracted discriminant is redundant with the already existing atoms. Property P_2 is also found to be a logical consequence of clause C_2:

```
                    sum([X,Y],R) ← add(X,Y,R)
                      DCI: C2     ↓     {}
[X,Y]=[?_|?_] & [X,Y]=[?HL|?TL] & sum(?TL,?TS) ← add(X,Y,R)
                  2 × DCI: LA(=)    ↓     {HL/X, TL/[Y]}
```

sum([Y],?TS) ← add(X,Y,R)
DCI: P_1 ↓ {*TS/Y*}
← add(X,Y,R)

The extracted clause is:

```
discSum2(L,S,HL,TL,TS) ←
                          L=[X,Y],S=R,
                          HL=X,TL=[Y],TS=Y,
                          add(X,Y,R)
```

However, the extracted clause defines *ProcComp*$_2$ rather than *Discriminate*$_2$. The generalization heuristics of Section 13.3.2 should thus not be used here, because they are tailored for the generalization of discriminating clauses. An adaptation of these heuristics to the generalization of processing and composition clauses is straightforward, and yields here:

```
pcSum2(HL,TS,S) ← add(HL,TS,S)
```

Hence *LA*$_6$*(sum)* is defined as follows (after some rewriting):

```
sum(L,S) ⇔
    L=[]       ∧ S=[]                                  {E1}
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ sum(TL,TS)
               ∧ add(HL,TS,S)                          {E2-E4}
```

There is no reason to re-invoke the Proofs-as-Programs Method at Step 7 (Synthesis of the *Discriminate*$_k$), because if there were a need for discriminants, they would have been synthesized at Step 6. Hence *LA*$_7$*(sum)* is equivalent to *LA*$_6$*(sum)*, which completes the synthesis of *LA(sum)*. ♦

14.3 A Methodology for Choosing "Good" Examples and Properties

In Section 6.2, we informally showed how to choose "good" properties, given a set of examples. The intuition was that such "good" properties are some form of disambiguating generalizations of the examples. More generally, an investigation of what constitutes a "good" example or a "good" property would lead to the formulation of a methodology for choosing such examples and properties. Following such a methodology should cut down on the amount of time spent on elaborating specifications, should possibly reduce to zero the amount of interaction between the specifier and the synthesis system, and should diminish the synthesis time, as less examples and properties have to be handled.

The formulation of such a methodology is of course influenced by the actual language for properties, and by the actual synthesis mechanism. We here stick to the corresponding decisions taken throughout Part III of this thesis: properties are here non-recursive Horn clauses whose heads involve the specified predicate.

There naturally is a nice analogy between such a methodology for choosing examples and the techniques of black-box test-data selection that are used in software validation. For instance, [Goodenough and Gerhart 75] suggest partitioning the domain of an algorithm into a finite number of equivalence classes (that are identified by the characteristics of the parameters of the algorithm), such that the correct behavior of the algorithm on one *representative* of each class will, "by induction", test the entire class, and hence establish the correctness of the algorithm. A more powerful technique is to choose *boundary-case* test-data of each equivalence class. For instance, rather than picking the empty list and any list of length >0 as representatives of a parameter of type *list*, one should pick three lists, of lengths 0, 1, and >1,

as its boundary-cases. But both techniques still suffer from the same drawback that they separately consider each characteristic of the parameters for the identification of the equivalence classes. So the idea is to consider all possible causality links between the characteristics of the parameters, and to actually construct a *decision table*, whose columns represent more meaningful equivalence classes, for which boundary-case test-data may then be picked.

The following is a first step towards such a methodology for choosing examples and properties. It is not formalized, and is thus reduced to a mere collection of informal guidelines that have been discovered through experimentation with the synthesis mechanism, as well as through our understanding of that mechanism. The methodology assists the specifier in the choice of a complete and minimal set of examples of the intended relation, and in the generalization of these examples into a complete and minimal set of properties of the intended relation $\mathcal{R}$:

(1) Select some inductively defined parameter (let's call it X) among the parameters of predicate r/n. (Note that the synthesis mechanism need not necessarily select the same parameter as the induction parameter.)

(2) Choose examples for each legal size of X up to some size m. Whenever possible, use different constants across the various examples. Make sure that the possible non-determinism of $\mathcal{R}$ (given a ground value of X) is apparent in the examples. The value of m may be determined by co-routining with the next step, but making sure that examples are chosen at least for all boundary cases.

(3) Generalize into properties, if possible, the examples where X is of a size equal to or less than some integer n, where n is the largest size whose examples lead to properties without recursion and without repetition of information in their bodies. Set m to $n+d$, where d is the decomposition decrement for X. The most useful generalization technique is the maximally repeated application of the replacing-a-constant-by-a-variable inductive inference rule to an example; this often requires a subsequent specialization by introduction of a body to the resulting unit clause.

This methodology abstracts away the internal apparatus of the synthesis mechanism, but algorithmic and mathematical concepts such as induction parameter, size, decomposition decrement, and so on, are hard to avoid in its formulation.

Note the interesting role of constants in examples: constants almost act like variables. More precisely, within an example, the use of different constants at different positions should reflect non-unifiability of the variables into which these constants could be turned. And the use of the same constant at different positions should reflect unifiability, if not identity, of the variables into which these constants could be turned. However, across several examples, the use of different constants at the same positions should reflect their value-irrelevance, and hence unifiability of the variables into which these constants could be turned. But unifiability (if not identity) and non-unifiability are not all there is to say about the variables into which the constants could be turned, because otherwise properties would have no different semantics than the examples they generalize. Indeed, properties have more disambiguating expressive power, in the sense that predicates other than $=/2$ and $\neq/2$ may be used in their bodies.

The methodology for choosing properties, as outlined here and in Section 6.2, shows that it is difficult to find "good" properties without actually having to reject recursive ones. Although being able to craft such recursive properties probably amounts to being able to design the final algorithm, recursive properties should thus be allowed in specifications, especially as an extreme case of incomplete specifications is a complete specification. Fortunately, our synthesis mechanism is able to handle recursive properties: we have only rejected them so as to show that they are not needed for successful algorithm synthesis, thus picking up a tougher challenge.

Given our perspective on formal specifications (see Section 1.2), we would like to call the act of choosing examples and properties an act of *programming by examples and properties*.

14.4 The SYNAPSE System

A prototype implementation of our synthesis mechanism is being developed: it is called SYNAPSE (short for *SYNthesis of logic Algorithms from PropertieS and Examples*), and is written in portable Prolog. In Section 14.4.1, we discuss the architecture of SYNAPSE, whereas Section 14.4.2 depicts some target scenarios of SYNAPSE at-work.

14.4.1 The Architecture of SYNAPSE

The architecture of SYNAPSE is very obvious from the technical details of the underlying synthesis mechanism (see Chapter 12 and Chapter 13): synthesis is achieved by a sequence of seven (mostly) non-deterministic procedures, each implementing one of the seven synthesis steps. The implementation is actually based on version 4 of the divide-and-conquer schema (page 114), unlike the theoretical presentation in this thesis, which is restricted to version 3. This means that arbitrary *n*-ary relations can be handled, rather than only binary ones.

The initial *input* to the synthesizer is a procedure declaration (consisting of the name of the specified predicate, and the names of the parameters), a set of examples, and a set of properties for that predicate. The theoretical aim of non-deterministically synthesizing all possible logic algorithms (by iterating over all possible induction parameters and decomposition predicates) is rather undesirable in practice, namely because the specifier might want to hint at personal preferences for these selections. Synthesis thus always stops at Step 2 (Synthesis of *Minimal* and *NonMinimal*) for letting the specifier select an induction parameter, its minimal form, and its non-minimal form. It also always stops at Step 3 (Synthesis of *Decompose*) for letting the specifier select a decomposition predicate. Of course, an entry of the corresponding database that hasn't been selected yet constitutes the default selection. For convenience, synthesis also always stops at Step 4 (Syntactic introduction of the recursive atoms), so that the specifier may declare which parameters are auxiliary parameters. A correct declaration indeed tremendously speeds up the computations at Step 6 (Synthesis of the $Process_k$ and the $Compose_k$). If the declaration is correct, but not complete (in the sense that some, if not all, auxiliary parameters are not declared as such), then synthesis proceeds correctly, but doesn't acquire the maximal possible speed-up. If the declaration is incorrect, then synthesis fails at some later step. The default is that only the parameters of non-inductively defined types are auxiliary parameters. All these optional specifier interventions may be cut off by switching from the default *manual mode* to the *automatic mode*.

The synthesis mechanism, as described in the previous two chapters, does not prescribe any mandatory dialogue with the specifier. But we concluded with the recommendation that any serious implementation of that mechanism should interact with the specifier whenever it applies one of the listed heuristics, or whenever all the mechanized methods yield unsatisfactory results on a specific task. The corresponding dialogues have however not yet been fully worked out (according to the guidelines of Section 11.1), and should be the object of future research for extending the current SYNAPSE implementation. Moreover, in order to quickly complete this first prototype implementation, several mechanizable computations have been replaced by queries to the specifier, although these interactions do not at all follow the mentioned guidelines. For instance, the computation (at Steps 4 to 6) of substitutions that render 3-tuples admissible is currently oracle-based, rather than automated. But these are restrictions of the implementation, not of the power of the underlying synthesis mechanism.

The *output* of the synthesizer is (besides the questions to the specifier) a possibly empty set of successfully designed logic algorithms. Indeed, the non-determinism of some synthesis steps makes the whole synthesis mechanism a non-deterministic mechanism. For debugging purposes, or for otherwise motivated inspections, SYNAPSE may even be switched to the *think-aloud mode*, where it prints out all the taken decisions, as well as all intermediate logic algorithms (of Steps 1 to 6). The default is the *quiet mode*. The printed-out logic algorithms are in canonical representation with respect to (version 4 of) the divide-and-conquer schema. Whereas the universal variables have the names given to them by the specifier in the procedure declaration (and are thus meaningful to the specifier), the existential variables have names invented by the synthesizer. However, the underlying divide-and-conquer strategy suggests a very useful naming scheme for these existential variables: the names of the heads (respectively, tails) of some parameter are prefixed by the letter "H" (respectively, "T"), and suffixed by a running number. The synthesized logic algorithms are thus actually in human-readable form, even though a machine-readable form would be sufficient in an ideal automatic programming setting where the output is only executed, but not inspected or debugged. Logic algorithms are not executable, but they are easily translated into, say, *Prolog* programs, so that they may be executed and thus tested. However, the synthesized logic algorithms are subject to many transformation and optimization opportunities before they should undergo such an implementation step: SYNAPSE is but a component of the software engineering chain, and should thus not be seen as a stand-alone system.

Particular care was taken to make SYNAPSE a clean meta-program (a program that manipulates or generates programs). Object-level variables (the variables of the designed logic algorithms) thus have a representation that is different from the one of meta-level variables (the variables of the synthesizer), even though this is a quite hard decision to stick to in most Prolog dialects. Indeed, the price to pay is the re-implementation of many *Prolog* features such as resolution, unification, the application and composition of substitutions, and so on, and hence a significant slowdown of synthesis. But the benefit would be that no use is made of *Prolog*'s "impure" predicates, such as *var*/1, ==/2, *call*/1, *assert*/1, *retract*/1, *clause*/2, …, and hence the existence of a clean semantics of the meta-program. The chosen representation of object-level variables is a ground representation, because a typed representation would be insufficient here, if not even harder to achieve. However, for reasons of speeding up the SYNAPSE implementation process, we have sometimes preferred to take a leave from this strict objective (namely by writing procedures that translate object-level formulas into meta-level formulas, so that predicates such as *call*/1 and *setof*/3 may be used). A better solution than fixing this would definitely be to re-implement the entire SYNAPSE system in a (logic) programming language that does have clean second-order programming facilities, such as *Gödel* [Hill and Lloyd 91].

The current SYNAPSE prototype features type inference for proposing default instances at Steps 2 and 3. It has fully automatic implementations of Step 1, of the ground case of the MSG Method (Step 6), and of the Proofs-as-Programs Method (Step 7). It provides semi-automatic implementations of Steps 4 and 5, and of the non-ground case of the MSG Method (Step 6).

14.4.2 Target Scenarios for SYNAPSE

We now list a few synthesis scenarios that SYNAPSE was targeted to reproduce. Pending the future extensions of the current prototype, these scenarios may not all already be obtained, but the synthesis mechanism is powerful enough to handle these problems. The used problems range from easy (*plateau*/3) to moderately difficult (*efface*/3) to intricate (*sort*/2). All the given syntheses are in *quiet* and *manual* mode.

Example 14-7: As a reminder, *plateau(N,E,P)* holds iff *P* is a plateau of *N* elements equal to *E*, where *N* is a positive integer, *E* a term, and *P* a non-empty list. The synthesis dialogue proceeds as follows (the actual dialogue is printed in the `Courier` font: synthesizer output precedes colons, whereas user input follows colons; the default answers to questions are between curly braces; comments on the synthesis are printed in the Times font):

```
Procedure declaration: plateau(N,E,P)
Example:  plateau(1,a,[a])
Example:  plateau(2,b,[b,b])
Example:  plateau(3,c,[,c,c,c])
Example:
Property: plateau(1,X,[X])
Property: plateau(2,Y,[Y,Y])
Property:
```

This ends the specification acquisition. Synthesis now starts with the expansion phase (Steps 1 to 4), and asks for the specifier's preferences (because the *manual* mode is *on*):

```
Induction parameter {N}: N
Minimal {N=s(0)}: N=s(0)
NonMinimal {N=s(s(_))}: N=s(s(_))
Decompose {N=s(TN) ∧ HN=N}: N=s(TN) ∧ HN=N
List of auxiliary parameters {[E]}: [E]
```

The specifier accepts all the default selections. Synthesis now goes fully automatically through the reduction phase (Steps 5 to 7), and yields:

```
LA(plateau) is
  plateau(N,E,P) ⇔
      N=s(0)     ∧ E=_ ∧ P=[E]
    ∨ N=s(s(_)) ∧ N=s(TN) ∧ HN=N
                 ∧ true
                 ∧ plateau(TN,E,TP)
                 ∧ P=[E|TP] ∧ HN=s(s(_)) ∧ TP=[_|_]
```

This logic algorithm is totally correct wrt the intended relation. Quite a few atoms can be simplified away (the last two, for instance). ♦

Example 14-8: As a reminder, *efface(E,L,R)* holds iff *R* is *L* without its first (existing) occurrence of *E*, where *E* is a term, *L* is a non-empty list, and *R* is a list. The synthesis dialogue proceeds as follows:

```
Procedure declaration: efface(E,L,R)
Example:  efface(a,[a],[])
Example:  efface(b,[b,c],[c])
Example:  efface(e,[d,e],[d])
Example:  efface(f,[f,g,h],[g,h])
Example:  efface(j,[i,j,k],[i,k])
Example:  efface(p,[m,n,p],[m,n])
Example:
Property: efface(X,[X],[])
Property: efface(X,[X,Y],[Y])
Property: efface(X,[Y,X],[Y]) ⇐ X≠Y
Property:
Induction parameter {L}: L
Minimal {L=[_]}: L=[_]
NonMinimal {L=[_,_|_]}: L=[_,_|_]
```

```
Decompose {L=[HL|TL]}: L=[HL|TL]
List of auxiliary parameters {[E]}: [E]
LA(efface) is
  efface(E,L,R) ⇔
      L=[_]       ∧ L=[HL] ∧ HL=E ∧ R=[]
    ∨ L=[_,_|_]   ∧ L=[HL|TL]
                  ∧ HL=E
                  ∧ R=TL
    ∨ L=[_,_|_]   ∧ L=[HL|TL]
                  ∧ HL≠E
                  ∧ efface(E,TL,TR)
                  ∧ R=[HL|TR]
```

This logic algorithm is totally correct wrt the intended relation. The main obstacle is the detection that recursion is useless in the second disjunct. ♦

Example 14-9: As a reminder, *sort(L,S)* holds iff *S* is an ascendingly ordered permutation of *L*, where *L, S* are integer lists. The synthesis dialogue proceeds as follows:

```
Procedure declaration: sort(L,S)
Example:  sort([],[])
Example:  sort([1],[1])
Example:  sort([2,3],[2,3])
Example:  sort([3,2],[2,3])
Example:  sort([4,5,6],[4,5,6])
Example:  sort([4,6,5],[4,5,6])
Example:  sort([5,4,6],[4,5,6])
Example:  sort([5,6,4],[4,5,6])
Example:  sort([6,4,5],[4,5,6])
Example:  sort([6,5,4],[4,5,6])
Example:
Property: sort([X],[X])
Property: sort([X,Y],[X,Y]) ⇐ X≤Y
Property: sort([X,Y],[Y,X]) ⇐ X>Y
Property:
Induction parameter {L}: L
Minimal {L=[]}: L=[]
NonMinimal {L=[_|_]}: L=[_|_]
Decompose {L=[HL|TL]}: L=[HL|TL]
List of auxiliary parameters {[]}: []
LA(sort) is
  sort(L,S) ⇔
      L=[]      ∧ S=[]
    ∨ L=[_|_]   ∧ L=[HL|TL]
                ∧ true
                ∧ sort(TL,TS)
                ∧ insert(HL,TS,S)
where
```

```
insert(E,L,R) ⇔
    L=[]       ∧ E=_ ∧ R=[E]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ HL≥E
               ∧ E=_ ∧ R=[E|L]
  ∨ L=[_|_]    ∧ L=[HL|TL]
               ∧ HL<E
               ∧ insert(E,TL,TR)
               ∧ R=[HL|TR]
```

This is the so-called *Insertion-Sort* algorithm. It is non-trivial in the sense that a new predicate, here for convenience called *insert*/3, needs to be invented from scratch. Moreover, this sub-synthesis needs to be done in totally automatic mode, as the specifier is not supposed to be able to answer questions about such invented predicates.

Synthesis then backtracks to the last choice-point (at Step 3 in this case), in its search for alternative logic algorithms:

Backtracking...

Decompose {firstPlateau(L,HL,TL)}:
L=[HL|T] ∧ partition(T,HL,TL_1,TL_2)

List of auxiliary parameters {[]}: []

LA(sort) is

```
sort(L,S) ⇔
    L=[]       ∧ S=[]
  ∨ L=[_|_]    ∧ L=[HL|T] ∧ partition(T,HL,TL1,TL2)
               ∧ true
               ∧ sort(TL1,TS1) ∧ sort(TL2,TS2)
               ∧ pcSort(HL,TS1,TS2,S)
```

where

```
pcSort(A,L1,L2,R) ⇔
    L1=[]      ∧ R=[A|L2]
  ∨ L1=[_|_]   ∧ L1=[HL1|TL1]
               ∧ true
               ∧ pcSort(A,TL1,TL2,TR)
               ∧ L2=TL2 ∧ R=[HL1|TR]
```

This is the so-called *Quick-Sort* algorithm. It is non-trivial in the sense that a new predicate, called *pcSort*/3, needs to be invented from scratch. It is however obvious that:

pcSort(A,L_1,L_2,R) ⇔ append(L_1,[A|L_2],R)

Again, this sub-synthesis needs to be done in totally automatic mode. Hence, the fact that L_2 actually also is an auxiliary parameter of *pcSort*/3, just like *A*, cannot be declared, and is only detected as such after this sub-synthesis has been completed.

Synthesis then backtracks to the last choice-point (again at Step 3 in this case), in its search for alternative logic algorithms:

Backtracking...

Decompose {firstPlateau(L,HL,TL)}: split(L,TL_1,TL_2)

List of auxiliary parameters {[]}: []

LA(sort) is

```
sort(L,S) ⇔
    L=[]       ∧ S=[]
  ∨ L=[_|_]    ∧ split(L,TL1,TL2)
               ∧ true
               ∧ sort(TL1,TS1) ∧ sort(TL2,TS2)
               ∧ merge(TS1,TS2,S)
where
  merge(A,B,C) ⇔
    C=[]       ∧ A=[] ∧ B=[]
  ∨ C=[_|_]    ∧ C=[HC|TC]
               ∧ (TC=[]) ∨ (TC=[HTC|_] ∧ HC≤HTC)
               ∧ merge(TA,TB,TC)
               ∧ A=TA ∧ B=[HC|TB]
  ∨ C=[_|_]    ∧ C=[HC|TC]
               ∧ (TC=[]) ∨ (TC=[HTC|_] ∧ HC>HTC)
               ∧ merge(TA,TB,TC)
               ∧ A=[HC|TA] ∧ B=TB
```

This is the so-called *Merge-Sort* algorithm. It is non-trivial in the sense that a new predicate, here conveniently called *merge*/3 rather than *pcSort*/3, needs to be invented from scratch. ♦

14.5 Evaluation

Our synthesis mechanism builds upon a wide variety of ideas found in algorithm design, inductive inference, deductive inference, theorem proving, and so on. Thus, as the sections on related work have shown, specifications by examples and properties (see Chapter 6), the use of algorithm schemas (see Chapter 8), and the use of theorem proving and constructive logics for inferring preconditions (see Chapter 9) are not necessarily new ideas (to algorithm synthesis). But they are here combined in a novel way, and this together with some other new results. The main contributions of this thesis are thus the formulation of a general framework for the stepwise synthesis of logic algorithms (see Chapter 7), the development of a new method for inferring (in a specific setting) non-recursive logic algorithms from examples (see Chapter 10), and the combination of all these ideas for the development of a schema-guided logic algorithm synthesis mechanism from examples and properties (see Chapter 12 and Chapter 13).

First, in Section 14.5.1, we provide a loose collection of insights into our synthesis mechanism, and then, in Section 14.5.2, we compare its performance to some related systems.

14.5.1 Insights

The planned use of our synthesis mechanism is for situations where only fragmentary information is available about the intended relation, or where the specifier is unwilling (if not unable) to come up with a more precise description of the intended relation. By the phrase "incomplete specifications", we thus mean specifications where some information about the intended relation is deliberately withheld, but where the synthesis expected to extrapolate that information. We thus exclude incomplete specifications in the sense of evolutionary specifications, where one also deliberately withholds some information, but without expecting any extrapolation. The idea there is to gradually refine an algorithm by adding functionalities to its specification, hence continuously upgrading the intentions.

Although our synthesis mechanism is non-incremental in its handling of the examples and properties, it may of course be used in an incremental fashion. This amounts to presenting

increasingly large specifications of the same intended relation to the synthesis mechanism. In our case, the latter always starts from scratch, for fear of missing a possibly more elegant or efficient algorithm by patching the previous one. However, incremental handling of examples and properties need not always amount to a patchwork approach, so this non-conservative attitude is not always justified. Replay information should thus be gathered during synthesis, so as to allow quicker decision-taking during a new synthesis.

With our specification approach, a single problem is specified, and a single logic algorithm is synthesized, at each time. The other predicates used in the specification are assumed to be primitives and to have correct logic algorithms available. This is an assumption that is not made systematically by the machine learning and inductive logic programming communities, where multiple related concepts may sometimes be learned simultaneously. Of course, the invention of new sub-problems and the synthesis of logic algorithms therefore (see Section 14.2.4) slightly alleviate this apparent drawback.

Even in programming-in-the-small, it is sometimes necessary to compose several small algorithms of different schemas before obtaining an algorithm for the overall problem. In this sense, the solution proposed by this thesis (namely focus on the divide-and-conquer schema) seems somewhat restricted. However, our vision of a synthesis system as a collection of several schema-dedicated systems or, even better, as a workbench with a schema-driven system coupled to a tool-box of schema-independent methods, shows a way of tackling this restriction. Moreover, there seems to be very little knowledge about how to automate such algorithm decomposition and composition anyway. So this aspect seems a likely candidate for human very-high-level intervention, and our synthesis mechanism should be seen as a component of such a global solution to algorithm synthesis. The here targeted class of algorithms, namely divide-and-conquer algorithms, is a very important and large class.

Another restriction of our synthesis mechanism is that no parameters (such as accumulators) can be added to the specified predicate, and that the types of parameters cannot be generalized. For instance, the *reverse(L,R)* relation cannot be generalized into the *reverse(L,R,A)* relation (where *R* is the concatenation of *A* and the reversed list of *L*), although this would lead to a more efficient algorithm than the so-called naive one. Or the *flatTree(T,L)* relation (where *L* is a list containing the elements of binary-tree *T* traversed in an infix manner) cannot be generalized into the *flatTree(Ts,L)* relation (where *L* is a list containing the concatenation of the flattened elements of the binary-tree-list *Ts*), although this would lead to a more efficient algorithm. These generalization processes are called *computational generalization* and *structural generalization*, respectively. The motivation behind such generalization processes is that, somewhat paradoxically, the generalized relation is often easier to implement, and that the resulting generalized algorithm is often more efficient on the initial relation than an algorithm directly designed for this initial relation. Our synthesis mechanism cannot perform such a generalization. However, it seems possible to acquire a specification by examples and properties of the generalized relation by (semi-)automatic derivation from the specification of the initial relation, using generalization schemas. See [Deville 90] for a discussion of such schemas, and for ways of automating the discovery of the needed "eureka".

An extreme case of incomplete specifications are complete specifications! And our synthesis mechanism is actually able to handle such complete specifications (such as properties with recursion). However, given a complete specification under the form of a correct definite logic program, our synthesis mechanism is only likely to rediscover the corresponding logic algorithm if the selected decomposition predicate is available to it. Pushing the idea of allowing (almost-)complete specifications further, one could even imagine using our synthesis mechanism for some form of "redundant (logic) programming", where the specifier provides examples plus any kind of properties (possibly including a correct definite program), and

where synthesis either yields an algorithm or detects an inconsistency within the specification (and is thus unable to design an algorithm).

The use of the logic programming framework, and hence its preference over other programming frameworks (such as the functional, object-oriented, or imperative paradigms), is justified by its enabling a semantic and syntactic continuity from the specification language (relational facts and Horn clauses) to the logic algorithm language (equivalence statements with disjunctive normal form formulas in the right-hand side) to the logic program language (definite clauses, in the case of Prolog). Moreover, especially the Proofs-as-Programs Method has a particularly elegant formulation in the logic programming framework.

14.5.2 Comparison with Related Systems

We have no empirical data (synthesis times, example consumption, ...) yet about a comparison between SYNAPSE and its related systems, such as MIS [Shapiro 82], CLINT [de Raedt and Bruynooghe 89], FOIL [Quinlan 90], or the other model-based systems of the ILP community (see Section 3.4), or the trace-based systems of the LISP community (see Section 3.3). But we may make some predictions by comparing the underlying synthesis mechanisms instead. This requires some analysis first.

The big advantage of ILP systems is their flexibility: literally any logic program can be learned[20] from examples alone. Sometimes, for instance in MIS, the price to pay is however the development of a new clause refinement operator for each new class of programs. The existing systems have greatly varying voraciousness in terms of example consumption, (human) oracle solicitation, and time complexity. But the main point is that their generality of scope is at the same time their limitation. Indeed, these learning mechanisms have no understanding of what they are learning, and the learning process thus often resembles a rather undisciplined patchwork approach. For instance, when MIS (equipped with the eager search strategy) synthesizes the Insertion-Sort algorithm (including the synthesis of the *insert*/3 predicate), a total of 238 clauses is reported by [Shapiro 82] to be tested before identifying the 5 correct ones. And it must be said that the search space is well-organized. More recent systems have of course improved upon these figures, but the search is still rather blind.

It also appears that these ILP learners are mostly used in what we would like to call the *teacher/learner setting*, where a teacher provides a learner with examples and (almost) exactly the background knowledge that is necessary for successful learning. This requires the teacher to know the targeted logic program, or to at least suspect which predicates might be useful (otherwise, s/he wouldn't be a teacher after all). For instance, in the above-mentioned synthesis of the Insertion-Sort algorithm, the choice of the examples seems to be very much guided by insights about what is wrong with the currently designed algorithm, rather than by what would be universally "good" examples, whatever the obtained sorting algorithm. Or, even worse, the synthesis of the Quick-Sort algorithm is often "forced" by the provision of the *partition*/4 predicate as the only background knowledge to the specification by examples for the sorting problem. The search space is thus pre-enumerated. However, this setting is not always realistic, and the reported performances of the developed learners are greatly biased by it. Indeed, these performances would simply crumble away in what we would like to call the *specifier/synthesizer setting*, where a specifier provides a synthesizer with any examples and with a lot of background knowledge. In this setting, the synthesizer is expected to come up with algorithms, as the specifier may have no, or little, ideas about how to do this (otherwise, s/he wouldn't use a synthesizer after all). Constructive induction (predicate invention) is then crucial, rather than an option. If the Quick-Sort algorithm is synthesized from a sort-

20. For ILP systems, we prefer "learning" over "synthesis", as we reserved the latter terminology for the learning of (recursive) algorithms/programs.

ing specification, that's great; and if it's "only" the Insertion-Sort algorithm, that's still fine. Ideally, a whole family of equivalent algorithms should however be synthesized from a given specification.

Although the ILP community has tremendously improved the quality of machine learning research (because the underlying logic framework is much cleaner than most custom-made frameworks, and because of the use of a very small set of inductive inference rules, rather than of a large arsenal of custom-made rules), we recommend breaking up ILP research into several domains. One of these domains is recursive logic program synthesis, and is the object of this dissertation. This domain is very important, and is, as shown by Table 3-1, sufficiently more specific than the general setting to deserve some very dedicated research. The objective of this separation of concerns is to get a tighter theoretical grip on the learning of the targeted class of concepts (relations in our case), and the introduction of more discipline into the learning process.

A few researchers have tackled the lack of discipline in the synthesis of recursive logic programs from examples: for instance, [Tinkham 90] and [Sterling and Kirschenbaum 91] investigate the use of schemas to guide synthesis. Curiously, the now virtually defunct research on trace-based synthesis of functional programs from examples [Summers 77] [Biermann 78] did not suffer from such a marked lack of discipline, even though this research preceded ILP research.

The idea of this thesis was to introduce discipline into the synthesis of recursive logic programs (the chosen sub-domain of ILP), and this by making knowledge available to the synthesizer in a well-structured way. This is clearly within the recent trend of cross-fertilizing empirical and analytical learning. First, algorithms knowledge is provided by means of an algorithm schema; this aspect is usually missing in ILP research. Second, more algorithms knowledge is made available by databases of useful instances for some of the predicate variables of a schema; this is roughly equivalent to the background knowledge used in ILP, but our structured approach doesn't suffer from a performance degradation in the specifier/synthesizer setting. Third, problem-related knowledge is given by the specifier under the form of properties, which are disambiguating generalizations of examples; this is again roughly equivalent to background knowledge, but our approach directly links this knowledge to the specified predicate, and thus allows the preservation of such associations.

As a conclusion, our synthesis mechanism has by construction a much smaller scope than the more general ILP systems (recursive logic programs rather than logic programs), but due to the thus possible usage of a very disciplined approach based on schema-guidance, and due to the availability of disambiguating properties, it is to be expected that our synthesis mechanism behaves better than ILP systems on the class of recursive logic programs. The improvement should be especially dramatic in the specifier/synthesizer setting. Considering that ILP systems are known to have an example consumption similar to trace-based systems (except for Biermann's system, which usually makes do with a single example) and to exhibit an improvement in terms of time complexity over these systems, it is by transitivity also to be expected that our synthesis mechanism improves upon these older systems as well. In other words: SYNAPSE does less, but it does it better!

Conclusion

In this thesis, we tackled the problem of logic algorithm synthesis from specifications by examples and properties, so as to illustrate our claim that program synthesis can be effectively performed by successively filling in the place-holders of some algorithm schema, each such instantiation being done by deploying the best-suited method of a generic tool-box of (inductive, deductive, …) methods.

Synthesis here started from incomplete specifications, in the sense that we deliberately admitted a lack of information in the specification with respect to the intentions: synthesis was meant to extrapolate these intentions. The developed specification formalism was a compromise between specifications by examples and axiomatic specifications, because examples of the intended relation were required, as well as properties (a specialized form of axioms, but also a generalized form of examples). These properties were added to disambiguate the examples. This specification approach was predicted to allow faster and more reliable synthesis than from examples alone, but also shown to allow more natural and understandable specifications than first-order logic axiomatizations.

The chosen programming paradigm was logic programming, but we were only interested in synthesizing recursive logic programs, which we called logic algorithms. Moreover, the focus was on the actual algorithm synthesis, and on the declarative semantics of such logic algorithms, but not at all on their transformation, optimization, or implementation.

In order to have a firm theoretical grip on the effectiveness of the synthesis, we developed a set of correctness criteria and algorithm comparison criteria, so as to have a framework for the elaboration of stepwise synthesis strategies. A particular such strategy, based on a non-incremental presentation of the examples and properties, was then discussed in detail.

The aim of decomposing algorithm synthesis into a succession of well-defined software engineering tasks brought us to the notion of algorithm schemas, because schema-guidance is a very powerful way to inject algorithm knowledge into synthesis, and hence to reduce search spaces. We exclusively focussed on the divide-and-conquer schema, and identified its place-holders.

We then suggested that synthesis need not be done by a unique mechanism in one pass, nor by a decomposition into sub-tasks that are all performed using the same kind of reasoning. Therefore, we developed a generic tool-box of methods for instantiating place-holders of algorithm schemas: one method performs inductive reasoning, another method performs deductive reasoning, yet another method simply retrieves instances from a database of known-to-be-useful instances. The input to these methods is (a part of) the specification as well as the algorithm synthesized so far.

We finally illustrated our claim of effective stepwise algorithm synthesis via schema-guidance and tool-box usage by finding a convenient mapping between the identified place-holders of the divide-and-conquer schema and the methods of the developed tool-box. This mapping is at the heart of our mechanism for synthesizing logic algorithms from specifications by examples and properties.

Compared to related systems, such as those of ILP (Inductive Logic Programming) research, our synthesis mechanism features a smaller scope (recursive logic programs rather than any logic programs), but also a much more disciplined approach, due to its schema-guidance, well-structured knowledge provision, augmented specifications, and use of the best-suited method of a tool-box for each sub-task. In other words, the pragmatic restriction of scope allowed us to achieve better results on the class of recursive logic programs.

Conventions

Here follows the complete list of the used typographic conventions, abbreviations, and predefined symbols.

Typographic conventions

Function symbols (functors) and predicate symbols start with a lower-case letter. We often write them out followed by a slash "/" and their arity. Examples are $a/0$, $f/2$, and $p/3$. Exceptions are some primitive symbols, such as $=/2$, $</2$, and $\bullet/2$. Functors of arity 0 are called *constants*. By abuse of language, we often say "predicate" instead of "predicate symbol". Variable symbols, function variable symbols, and predicate variable symbols start with an upper case letter. Examples are X, F, and P. An anonymous variable is denoted by an underscore "_". Schema variable symbols and notation variable symbols consist of a lower-case letter. Examples are i, j, k, m, and n. The distinction with one-letter constants should always be obvious from context. By abuse of language, we often say "variable" instead of "variable symbol".

Terms and atoms are usually written in prefix notation. If no ambiguity arises, unary terms and atoms are sometimes written without parentheses, while binary terms and atoms are sometimes written in infix notation.

A constructed list with head H and tail T is denoted $H\bullet T$, whereas *nil* denotes the empty list. Another notation for a constructed list with head H and tail T is $[H|T]$, whereas [] is another notation for the empty list. These alternative notations allow shorthands such as $[H_1,H_2,\ldots,H_n]$ for $\bullet(H_1,\bullet(H_2,\bullet(\ldots,\bullet(H_n,nil)\ldots)))$, and $[H_1,H_2,\ldots,H_n|T]$ for $\bullet(H_1,\bullet(H_2,\bullet(\ldots,\bullet(H_n,T)\ldots)))$, where $n>0$.

Non-negative integers are successors of the constant zero, which is denoted 0. The sequence of integers is 0, $s(0)$, $s(s(0))$, …, $s^n(0)$, …, where $s/1$ is called the successor functor. Shorthands are $0, 1, 2, \ldots, n, \ldots$, respectively.

Variable symbols, predicate variable symbols, functors, and predicate symbols within text paragraphs are written in *Times-italics*. However, entirely formalized paragraphs, such as specifications, (logic) algorithms, and (logic) programs, are written in `Courier`.

Term vectors and variable vectors of indeterminate, but finite, length are denoted by **boldface** symbols. Examples are $\boldsymbol{t}$ and $\boldsymbol{X}$. Vectors of vectors of indeterminate, but finite, length are denoted by **underlined boldface** symbols. Examples are $\underline{\boldsymbol{t}}$ and $\underline{\boldsymbol{X}}$. Given an integer n, an n-tuple of length n is written using angled brackets. An example is $\langle t_1,t_2,\ldots,t_n\rangle$. Note that an n-tuple is a term, whereas a vector is not a term.

Names of sets or relations start with an upper-case letter, and are written in *Zapf*. An example is $\mathcal{R}$. Exceptions are some primitive symbols, such as $=/2$, $</2$, and $\in/2$. The predicate symbol corresponding to a relation is the name of the relation, but it then starts with a lower-case letter, and is written in *Times-italics*. For instance, r is the predicate symbol for relation $\mathcal{R}$. The complement of a relation has the same name as the relation itself, but crossed out by a slash "/", if the name is a primitive symbol, and $\overline{\text{overlined}}$, otherwise.

The binding of a term t to a variable X is denoted X/t. Substitutions are denoted by Greek lower-case letters. Examples are σ, ρ, and θ.

The construct $F[\boldsymbol{X}]$ denotes a well-formed formula F whose free variables are $\boldsymbol{X}$. The construct $F[\boldsymbol{t}]$ then denotes $F[\boldsymbol{X}]$ where the free occurrences of $\boldsymbol{X}$ are replaced by the terms $\boldsymbol{t}$. The **boldface** construct $\boldsymbol{r(s,t)}$ denotes a finite conjunction $r(s_1,t_1) \wedge r(s_2,t_2) \wedge \ldots \wedge r(s_n,t_n)$.

The end of a multi-paragraph example is indicated by a black diamond: ♦.

The end of a proof is indicated by a quad: ❑.

Emphasized words are underlined. Defined words are in *italics*.

Abbreviations

General abbreviations

iff	if and only if
N/A	non-applicable
wrt	with respect to
v(n):p–q	volume *v*, number *n*, pages *p–q*

Scientific abbreviations

AI	Artificial Intelligence
BNF	Backus-Naur Form
DCI	Definite Clause Inference
EBG	Explanation-Based Generalization
EBL	Explanation-Based Learning
gci	greatest common instance
glb	greatest lower bound
ILP	Inductive Logic Programming
LA	Logic Algorithm
LP	Logic Program
lub	least upper bound
mgu	most general unifier
ML	Machine Learning
msg	most specific generalization
NF	Negation-as-Failure
NFI	Negation-as-Failure Inference
Sim	Simplification inference
SL resolution	Linear resolution with Selection Function
SLD resolution	SL resolution for Definite clauses
SLDNF resolution	SLD resolution with the NF rule
Spec	specification
SYNAPSE	SYNthesis of logic Algorithms from PropertieS and Examples
wff	well-formed formula
wfr	well-founded relation

Glossary of Symbols

Sets

$\mathcal{A}$	the set of atoms constructed from $\mathcal{Q}$ and $\mathcal{T}$
$\mathcal{B}$	the Herbrand base (that is the set of ground atoms constructed from $\mathcal{Q}$ and $\mathcal{U}$); $\mathcal{B} \subseteq \mathcal{A}$
$\mathcal{C}$	the set of predefined base case constants of inductively defined data types; $\mathcal{C} \subseteq \mathcal{F}$; $\mathcal{C}$ is here assumed to be $\{0, nil\}$
$\mathcal{E}(r)$	a set of examples of predicate r
$\mathcal{F}$	the set of functors
$\mathcal{G}(r)$	a set of general examples of predicate r
$\mathcal{I}$	the intended interpretation
I, J, K, L	index sets
$\mathcal{L}(r)$	the set of logic algorithms for predicate r; $\mathcal{L}(r) \subseteq \mathcal{W}$
$\mathcal{M}(r)$	the set of possibly infinite logic algorithms for predicate r
$\mathcal{P}(r)$	a set of properties of predicate r
$\mathcal{Q}$	the set of predicate symbols ($\mathcal{F} \neq \mathcal{Q}$)

$\mathcal{R}$	the intended relation
$\mathcal{T}$	the set of terms constructed from $\mathcal{F}$ and $\mathcal{V}$
$\mathcal{U}$	the Herbrand universe (that is the set of ground terms constructed from $\mathcal{F}$); $\mathcal{U} \subseteq \mathcal{T}$
$\mathcal{V}$	the set of variable symbols
$\mathcal{W}$	the set of wff constructed from $\mathcal{A}$ and $\mathcal{V}$

Constants

f	the number of steps of a synthesis mechanism
m	the number of (general) examples: $m = \#\mathcal{E}(r)$, or $m = \#\mathcal{G}(r)$
nil or []	the empty list
p	the number of properties: $p = \#\mathcal{P}(r)$
ω	infinity
0	the integer zero
$\varnothing$ or {}	the empty set

Functors of arity n, where $n>0$

cons(E)	the set of constants occurring in expression E
dom(r)	the domain of a procedure for predicate r: $dom(r) \subseteq \mathcal{U}$
$dom(\sigma)$	the domain of substitution σ: $dom(\sigma) \subseteq \mathcal{V}$
funct(E)	the set of functors occurring in expression E
msg(s,t)	the most specific generalization of terms s and t
$\mathcal{P}(\mathcal{S})$	the set of subsets of set $\mathcal{S}$
$range(\sigma)$	the range of a substitution σ: $range(\sigma) \subseteq \mathcal{T}$
s(i)	the successor of integer i
vars(E)	the set of variables occurring in expression E
$\#\mathcal{S}$	the number of elements of set $\mathcal{S}$, or of vector $\mathcal{S}$
$H \bullet T$	the list constructed of head H and tail T
$\mathcal{S}_1 \cup \mathcal{S}_2$	the union of the sets $\mathcal{S}_1$ and $\mathcal{S}_2$
$\mathcal{S}_1 \cap \mathcal{S}_2$	the intersection of the sets $\mathcal{S}_1$ and $\mathcal{S}_2$
$\mathcal{S}_1 \setminus \mathcal{S}_2$	the difference of the sets $\mathcal{S}_1$ and $\mathcal{S}_2$

First-order variables of (version 3 of) the divide-and-conquer schema

X	the induction parameter
Y	the other parameter
$\boldsymbol{HX}$	the heads of X
$\boldsymbol{HY}$	the heads of Y
$\boldsymbol{TX}$	the tails of X
$\boldsymbol{TY}$	the tails of Y

Schema variables of (version 3 of) the divide-and-conquer schema

c	the number of sub-cases of the non-minimal case: $c = v + w$
d	the decomposition decrement: $d \geq 1$
h	the number of heads of X: $h = \#\boldsymbol{HX}$
h'	the number of heads of Y: $h' = \#\boldsymbol{HY}$
n	the arity of r
t	the number of tails of X and Y: $t = \#\boldsymbol{TX} = \#\boldsymbol{TY}$
u	the position of the induction parameter within an atom of r/n
v	the number of sub-cases of the non-recursive case
w	the number of sub-cases of the recursive case

Primitive predicates

false/0	never holds
true/0	always holds

$s = t$	term s is unifiable with term t
$s \leq t$	term s is less general than term t, or integer s is less than or equal to integer t (according to context)
$e \in \mathcal{S}$	term e is an element of set $\mathcal{S}$
$\mathcal{S}_1 \subseteq \mathcal{S}_2$	set $\mathcal{S}_1$ is a subset of set $\mathcal{S}_2$

Connectives for logic algorithms, logic algorithm schemas, properties, and implicative goals

$\forall$	for all (universal quantification)
$\exists$	there is (existential quantification)
$\neg$	not (negation)
$\vee$	inclusive or
$\veebar$	exclusive or
$\wedge$	and
$\Rightarrow$	implies
$\Leftarrow$	if
$\Leftrightarrow$	if-and-only-if
$\vee_{a \leq i \leq b} F_i$	$F_a \vee F_{a+1} \vee \ldots \vee F_b$, if $b \geq a$, and *false* otherwise
$\wedge_{a \leq i \leq b} F_i$	$F_a \wedge F_{a+1} \wedge \ldots \wedge F_b$, if $b \geq a$, and *true* otherwise

More conventions for implicative goals

❑	the empty implicative goal
?X	an undecided variable X
Y	a free variable Y

Connectives for logic programs

,	and
$\leftarrow$	if

Logic algorithms

$\bot_r, \top_r$	bottom, top of the logic algorithm lattice $(\mathcal{M}(r), \leq)$
$\leq, \geq, \cong$	is semantically less general than, more general than, equivalent to
«, », $\approx$	is syntactically less general than, more general than, equivalent to
exp/3	expansion function, relative an example set and an oracle
gen/2	syntactic generalization relation
spec/2	syntactic specialization relation

Meta-logical connectives

$\models$	Herbrand-logical consequence
$\vdash$	derivability

Precedence hierarchy (highest-to-lowest) of the wff connectives

$\neg, \forall, \exists$
$\vee, \veebar$
$\wedge$
$\Leftarrow, \Rightarrow, \Leftrightarrow$

References

AAAI = American Association for Artificial Intelligence
AII = Analogical and Inductive Inference
ALT = Algorithmic Learning Theory
CADE = Conference on Automated DEduction
ICLP = International Conference on Logic Programming
ILP = Inductive Logic Programming
IJCAI = International Joint Conference on Artificial Intelligence
JICSLP = Joint International Conference and Symposium on Logic Programming
LOPSTR = International Workshop on LOgic Program Synthesis and TRansformation
NACLP = North American Conference on Logic Programming
PLILP = Int'l Symp. on Programming Language Implementation and Logic Programming
SLP = International Symposium on Logic Programming

ACM = Association for Computing Machinery
IEEE = Institute of Electrical and Electronics Engineers
IFIP = International Federation for Information Processing
LNCS = Lecture Notes in Computer Science
MIT = Massachusetts Institute of Technology

[Angluin 84]
Dana Angluin. Inductive inference: Efficient algorithms. In [Biermann *et al*. 84a], pages 503–515.

[Angluin and Smith 83]
Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *Computing Surveys* 15(3):237–269, September 1983.

[Balzer 85]
Robert Balzer. A 15 year perspective on automatic programming. In [IEEE-TSE 85], pages 1257–1268.

[Balzer *et al*. 77]
Robert Balzer, Neil Goldman, and David Wile. Informality in program specifications. In *Proceedings of IJCAI'*77, pages 389–397. Also in *IEEE Transactions on Software Engineering* 4(2):94–102, March 1978. Also in [Rich and Waters 86a], pages 223–232.

[Banerji 84]
Ranan B. Banerji. Some insights into automatic programming using a pattern recognition viewpoint. In [Biermann *et al*. 84a], pages 483–502.

[Barker-Plummer 90]
Dave Barker-Plummer. Cliché programming in Prolog. In M. Bruynooghe (editor), *Proceedings of Meta'*90, pages 247–256.

[Barr and Feigenbaum 82]
Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*. Morgan Kaufmann, 1982.

[Barstow 77]
David R. Barstow. A knowledge-based system for automatic program construction. In *Proceedings of IJCAI'*77, pages 382–388.

[Barstow 79a]
David R. Barstow. The roles of knowledge and deduction in program synthesis. In *Proceedings of IJCAI'*79, pages 37–43. Early version of [Barstow 84a].

[Barstow 79b]
David R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence* 12(2):73–119, August 1979. Also in [Webber and Nilsson 81], pages 289–312. Also in [Rich and Waters 86a], pages 133–156.

[Barstow 84a]
David R. Barstow. The roles of knowledge and deduction in algorithm design. In [Biermann *et al*. 84a], pages 201–222.

[Barstow 84b]
David R. Barstow. A perspective on automatic programming. *AI Magazine* Spring 1984:5–27. Also in [Rich and Waters 86a], pages 537–559.

[Barstow 85]
David R. Barstow. Domain-specific automatic programming. In [IEEE-TSE 85], pages 1321–1336.

[Bates and Constable 85]
Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems* 7(1):113–136, January 1985.

[Bauer 79]
Michael A. Bauer. Programming by examples. *Artificial Intelligence* 12(1):1–21, May 1979. Also in [Rich and Waters 86a], pages 317–327.

[Bauer *et al*. 89]
Friedrich L. Bauer, Bernhard Möller, Helmut Partsch, and Peter Pepper. Formal program construction by transformations: Computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering* 15(2):165–180, February 1989.

[Beeson 85]
Michael J. Beeson. *Foundations of Constructive Mathematics*. A Series of Modern Surveys in Mathematics, Volume 6, Springer-Verlag, 1985.

[Bibel 80]
Wolfgang Bibel. Syntax-directed, semantics-supported program synthesis. *Artificial Intelligence* 14(3):243–261, October 1980.

[Bibel and Hörnig 84]
Wolfgang Bibel and K. M. Hörnig. LOPS: A system based on a strategical approach to program synthesis. In [Biermann *et al*. 84a], pages 69–89.

[Bibel and Biermann 93]
Wolfgang Bibel and Alan W. Biermann (editors). *Journal of Symbolic Computation: Special Issue on Automatic Programming*, 1993. (To appear.)

[Biermann 72]
Alan W. Biermann. On the inference of Turing machines from sample computations. *Artificial Intelligence* 3(3):181–198, Fall 1972.

[Biermann 78]
Alan W. Biermann. The inference of regular LISP programs from examples. *IEEE Transactions on Systems, Man, and Cybernetics* 8(8):585–600, 1978.

[Biermann 83]
Alan W. Biermann. Natural language programming. In [Biermann and Guiho 83], pages 335–368.

[Biermann 84a]
Alan W. Biermann. Dealing with search. In [Biermann *et al*. 84a], pages 375–392.

[Biermann 84b]
Alan W. Biermann. Some examples of program synthesis. In [Biermann *et al*. 84a], pages 553–561.

[Biermann 86]
Alan W. Biermann. Fundamental mechanisms in machine learning and inductive inference. In W. Bibel and Ph. Jorrand (editors), *Fundamentals of Artificial Intelligence*, pages 133–169. LNCS 232, Springer-Verlag, 1986.

[Biermann 92]
Alan W. Biermann. Automatic programming. In S. C. Shapiro (editor), *Encyclopedia of Artificial Intelligence*, pages 59–83. John Wiley, 1992. Second, extended edition.

[Biermann and Krishnaswamy 76]
Alan W. Biermann and Ramachandran Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering* 2(3):141–153, September 1976.

[Biermann and Smith 79]
Alan W. Biermann and Douglas R. Smith. A production rule mechanism for generating LISP code. *IEEE Transactions on Systems, Man, and Cybernetics* 9(5):260–276, May 1979.

[Biermann and Guiho 83]
Alan W. Biermann and Gérard Guiho (editors). *Computer Program Synthesis Methodologies*. Volume ASI-C95, D. Reidel, Dordrecht (NL), 1983.

[Biermann *et al*. 84a]
Alan W. Biermann, Gérard Guiho, and Yves Kodratoff (editors). *Automatic Program Construction Techniques*. Macmillan, 1984.

[Biermann *et al*. 84b]
Alan W. Biermann, Gérard Guiho, and Yves Kodratoff. An overview of automatic program construction techniques. In [Biermann *et al*. 84a], pages 3–30.

[Biggerstaff 84]
Ted J. Biggerstaff. Design directed synthesis of LISP programs. In [Biermann *et al*. 84a], pages 393–420.

[Boyer and Moore 79]
Robert S. Boyer and J. S. Moore. *A Computational Logic*. ACM Monograph Series, Academic Press, 1979.

[Bratko and Grobelnik 93]
Ivan Bratko and Marko Grobelnik. Inductive learning applied to program construction and verification. In *Proceedings of ILP'*93 (Technical Report IJS-DP-6707, Jozef Stefan Institute, Ljubljana, Slovenia), pages 279–292.

[Broy 83]
Manfred Broy. Program construction by transformations: A family tree of sorting programs. In [Biermann and Guiho 83], pages 1–49.

[Bruynooghe and De Schreye 89]
Maurice Bruynooghe and Danny De Schreye. Some thoughts on the role of examples in program transformation and its relevance for explanation-based learning. In K. P. Jantke (editor), *Proceedings of AII'*89, pages 60–77. LNCS 397, Springer-Verlag, 1989.

[Bundy 88]
Alan Bundy. A broader interpretation of logic in logic programming. In R. A. Kowalski and K. A. Bowen (editors), *Proceedings of ICLP'*88, pages 1624–1648. MIT Press.

[Bundy *et al*. 90]
Alan Bundy, Alan Smaill, and Geraint Wiggins. The synthesis of logic programs from inductive proofs. In J. W. Lloyd (editor), *Proceedings of the ESPRIT Symposium on Computational Logic*, pages 135–149. Springer-Verlag, 1990.

[Bundy *et al*. 93]
Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence* 1993. (To appear.)

[Buntine 88]
Wray Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence* 36(2):149–176, September 1988.

[Burstall and Darlington 77]
Rodney M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM* 24(1):44–67, January 1977.

[Clark 78]
Keith L. Clark. Negation-as-failure. In H. Gallaire and J. Minker (editors), *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[Clark 79]
Keith L. Clark. *Predicate logic as a computational formalism*. Technical Report DOC-79/59, Imperial College, London (UK), 1979.

[Clark 81]
Keith L. Clark. *The synthesis and verification of logic programs*. Technical Report DOC-81/36, Imperial College, London (UK), September 1981.

[Clark and Sickel 77]
Keith L. Clark and Sharon Sickel. Predicate logic: A calculus for deriving programs. In *Proceedings of IJCAI'*77, pages 410–411.

[Clark and Tärnlund 77]
Keith L. Clark and Sten-Åke Tärnlund. A first order theory of data and programs. In *Proceedings of the* 1977 *IFIP Congress*, pages 939–944. North-Holland.

[Clark and Darlington 80]
Keith L. Clark and John Darlington. Algorithm classification through synthesis. *The Computer Journal* 23(1):61–65, 1980.

[Clement and Lau 92]
Tim Clement and Kung-Kiu Lau (editors), *Proceedings of LOPSTR'*91. Workshops in Computing Series, Springer-Verlag, 1992.

[Cohen and Sammut 84]
Brian Cohen and Claude Sammut. Program synthesis through concept learning. In [Biermann *et al*. 84a], pages 463–482.

[Constable *et al*. 86]
Robert L. Constable, Stuart F. Allen, H. M. Bromley, *et al*. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[Coquand and Huet 86]
Thierry Coquand and Gérard Huet. *The calculus of construction*. Rapport de recherche No. 530, INRIA-Paris (France), 1986.

[Cormen *et al*. 90]
Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms*. MIT Press, Electrical Engineering and Computer Science Series, 1990.

[Darlington 81]
John Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence* 16(1):1–46, March 1981. Also in [Rich and Waters 86a], pages 99–121.

[Darlington 83]
John Darlington. The synthesis of implementations for abstract data types. In [Biermann and Guiho 83], pages 309–334.

[De Boeck and Le Charlier 90]
Pierre De Boeck and Baudouin Le Charlier. Static type analysis of Prolog procedures for ensuring correctness. In P. Deransart and J. Maluszynski (editors), *Proceedings of PLILP'*90, pages 222–237. LNCS 456, Springer-Verlag.

[de Bruijn 80]
N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley (editors), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[DeJong and Mooney 86]
Gerald F. DeJong and Raymond Mooney. Explanation-based learning: An alternative view. *Machine Learning* 1(2):145–176, 1986.

[De Raedt and Bruynooghe 88]
Luc De Raedt and Maurice Bruynooghe. On interactive concept-learning and assimilation. In *Proceedings of the 3rd European Working Session on Learning*, pages 167–176. Pitman (Glasgow), UK, 1988.

[De Raedt and Bruynooghe 89]
Luc De Raedt and Maurice Bruynooghe. Towards friendly concept-learners. In *Proceedings of IJCAI'*89, pages 849–856.

[De Raedt and Bruynooghe 90]
Luc De Raedt and Maurice Bruynooghe. Indirect relevance and bias in inductive concept learning. *Knowledge Acquisition* 2:365–390, 1990.

[De Raedt and Bruynooghe 92]
Luc De Raedt and Maurice Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence* 53(2–3):291–307, February 1992.

[Dershowitz and Lee 87]
Nachum Dershowitz and Yuh-Jeng Lee. Deductive debugging. In *Proceedings of SLP'*87, pages 298–306.

[Deville 87]
Yves Deville. *A Methodology for Logic Program Construction*. PhD Thesis, Facultés Universitaires Notre-Dame de la Paix, Namur (Belgium), 1988.

[Deville 88]
Yves Deville. *Generalized Herbrand interpretations*. Technical Report 88-21, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Namur (Belgium), 1988.

[Deville 90]
Yves Deville. *Logic Programming: Systematic Program Development*. International Series in Logic Programming, Addison Wesley, 1990.

[Deville 94]
Yves Deville (editor), *Proceedings of LOPSTR'*93. Workshops in Computing Series, Springer-Verlag, 1994. (To appear.)

[Deville and Burnay 89]
Yves Deville and Jean Burnay. Generalization and program schemata: A step towards computer-aided construction of logic programs. In E. L. Lusk and R. A. Overbeek (editors), *Proceedings of NACLP'*89, pages 409–425. MIT Press.

[Deville and Flener 93]
Yves Deville and Pierre Flener. *Correctness criteria for logic program synthesis*. Research Report, Unité d'Informatique, Université Catholique de Louvain, Louvain-la-Neuve (Belgium), 1993. (In preparation.)

[Deville and Lau 94]
Yves Deville and Kung-Kiu Lau. Logic program synthesis: A survey. *Journal of programming, Special Issue on* 10 *Years of Logic Programming*, 1994. (To appear.)

[Drabent *et al*. 88]
Wlodek Drabent, Simin Nadjm-Tehrani, and Jan Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M. H. Rogers (editors), *Meta-Programming in Logic Programming: Proceedings of Meta'*88, pages 501–521. MIT Press.

[Eriksson 84]
Lars-Henrik Eriksson. Synthesis of a unification algorithm in a logic programming calculus. *Journal of Logic Programming* 1(1):3-33, June 1984.

[Eriksson and Johansson 81]
Agneta Eriksson and Anna-Lena Johansson. *NATDED: A derivation editor*. Technical Report UPMAIL-3, Uppsala (Sweden), 1981.

[Eriksson and Johansson 82]
Agneta Eriksson and Anna-Lena Johansson. Computer-based synthesis of logic programs. In M. Dezani-Ciancaglini and U. Montanari (editors), *Proceedings of an International Symposium on Programming*, pages 105–115. LNCS 137, Springer-Verlag, 1982.

[Eriksson *et al*. 83]
Agneta Eriksson, Anna-Lena Johansson, and Sten-Åke Tärnlund. Towards a derivation editor. In M. van Caneghem and D. H. D. Warren (editors), *Logic Programming and its Applications*, pages 117–126. Ablex, 1986.

[Feather 87]
Martin S. Feather. A survey and classification of some program transformation approaches and techniques. In L. G. L. T. Meertens (editor), *Program Specification and Transformation*, *Proceedings of the* 1987 *IFIP Congress*, pages 165–195. Elsevier Science Publishers (North-Holland), 1987.

[Fickas 85]
Stephen F. Fickas. Automating the transformational development of software. In [IEEE-TSE 85], pages 1268–1277.

[Flener 89]
Pierre Flener. *(Semi-)automatic software development: State of the art*. Folon Research Paper, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Namur (Belgium), 1989.

[Flener 91]
Pierre Flener. *Towards programming by examples and properties*. Technical Report CS-1991-09, Duke University, Durham (NC, USA), 1991.

[Flener and Deville 91a]
Pierre Flener and Yves Deville. Synthesis of composition and discrimination operators for divide-and-conquer logic programs. In [Jacquet 93], pages 67–96.

[Flener and Deville 91b]
Pierre Flener and Yves Deville. Towards stepwise, schema-guided synthesis of logic programs. In [Clement and Lau 92], pages 46–64.

[Flener and Deville 92]
Pierre Flener and Yves Deville. SYNAPSE, a system for logic program synthesis from incomplete specifications. In M. Ducassé, Y.-J. Lin, and Ü. Yalçinalp (editors), *Proceedings of the JICSLP'92 Workshop on Logic Programming Environments*, pages 9–15. Available as Technical Report CAISR-92-143, Case Western Reserve University, Cleveland (OH, USA).

[Flener and Deville 93]
Pierre Flener and Yves Deville. Logic program synthesis from incomplete specifications. Accepted for publication in [Bibel and Biermann 93].

[Follet 84]
Ria Follet. Combining program synthesis with program analysis. In [Biermann *et al*. 84a], pages 91–107.

[Fribourg 90]
Laurent Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In D. H. D. Warren and P. Szeredi (editors), *Proceedings of ICLP'90*, pages 685–699. MIT Press. Updated and revised version in [Fribourg 91].

[Fribourg 91a]
Laurent Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In [Jacquet 93], pages 39–66.

[Fribourg 91b]
Laurent Fribourg. Automatic generation of simplification lemmas for inductive proofs. In V. Saraswat and K. Ueda (editors), *Proceedings of SLP'91*, pages 103–116. MIT Press.

[Garey and Johnson 79]
Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[Gegg-Harrison 89]
Timothy S. Gegg-Harrison. *Basic Prolog schemata*. Technical Report CS-1989-20, Duke University, Durham (NC, USA), 1989.

[Gegg-Harrison 93]
Timothy S. Gegg-Harrison. *Exploiting Program Schemata in a Prolog Tutoring System*. PhD Thesis, Duke University, Durham (NC, USA), 1993.

[Genesereth and Nilsson 88]
Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, 1988.

[Gold 67]
E. Mark Gold. Language identification in the limit. *Information and Control* 10(5):447–474, 1967.

[Goldberg 86]
Allen T. Goldberg. Knowledge-based programming: A survey of program design and construction techniques. *IEEE Transactions on Software Engineering* 12(7):752–768, July 1986.

[Goodenough and Gerhart 75]
John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering* 1(2):156–173, June 1975.

[Green 69]
Cordell Green. Application of theorem proving to problem solving. In *Proceedings of IJCAI'*69, pages 219–239. Also in [Webber and Nilsson 81], pages 202–222.

[Green 77]
Cordell Green. A summary of the PSI program synthesis system. In *Proceedings of IJCAI'*77, pages 380–381.

[Green and Barstow 78]
Cordell Green and David R. Barstow. On program synthesis knowledge. *Artificial Intelligence* 10(3):241–270, November 1978. Also in [Rich and Waters 86a], pages 455–474.

[Green *et al*. 79]
Cordell Green *et al*. Results in knowledge-based program synthesis. In *Proceedings of IJCAI'*79, pages 342–344.

[Green and Westfold 82]
Cordell Green and Stephen Westfold. Knowledge-based programming self applied. *Machine Intelligence* 10, 1982. John Wiley. Also in [Rich and Waters 86a], pages 259–284.

[Green *et al*. 83]
Cordell Green, David Luckham, Robert Balzer, Thomas Cheatham, and Charles Rich. Report on a knowledge-based software assistant. In [Rich and Waters 86a], pages 377–428.

[Gupta 87]
Ajay Gupta. Explanation-based failure recovery. In *Proceedings of AAAI'*87, Volume 2, pages 606–610. Morgan Kaufmann.

[Hagiya 90]
Masami Hagiya. Programming by example and proving by example using higher-order unification. In M. E. Stickel (editor), *Proceedings of CADE'*90, pages 588–602. LNCS 449, Springer-Verlag.

[Hansson 80]
Åke Hansson. *A Formal Development of Programs*. PhD Thesis, University of Stockholm (Sweden), 1980.

[Hansson and Tärnlund 79]
Åke Hansson and Sten-Åke Tärnlund. A natural programming calculus. In *Proceedings of IJCAI'*79, pages 348-355.

[Hansson *et al*. 82]
Åke Hansson, Seif Haridi, and Sten-Åke Tärnlund. Properties of a logic programming language. In K. L. Clark and S.-Å. Tärnlund (editors), *Logic Programming*, pages 267–280. Academic Press, 1982.

[Hardy 75]
Steven Hardy. Synthesis of LISP functions from examples. In *Proceedings of IJCAI'*75, pages 240–245.

[Hayashi 87]
Susumu Hayashi. PX: A system extracting programs from proofs. In M. Wirsing (editor), *Formal Description of Programming Concepts*, pages 399–423. Elsevier Science Publishers B.V. (North Holland), 1987.

[Heidorn 76]
George E. Heidorn. Automatic programming through natural language dialogue: A survey. *IBM Journal on Research and Development* 20(4):302–313, 1976. Also in [Rich and Waters 86a], pages 203–214.

[Henrard and Le Charlier 92]
Jean Henrard and Baudouin Le Charlier. FOLON: An environment for declarative construction of logic programs. In M. Bruynooghe and M. Wirsing (editors), *Proceedings of PLILP'92*, pages 217–231. LNCS 631, Springer-Verlag.

[Hill and Lloyd 91]
Patricia M. Hill and John W. Lloyd. *The Gödel Report*. Technical Report 91-02, Department of Computer Science, University of Bristol, 1991.

[Hogger 78]
Christopher J. Hogger. Program synthesis in predicate logic. In D. Sleeman (editor) *Proceedings of the AISB/GI Conference on Artificial Intelligence* (Hamburg, Germany), pages 22–27. 1978.

[Hogger 81]
Christopher J. Hogger. Derivation of logic programs. *Journal of the ACM* 28(2):372–392, April 1981.

[Hogger 84]
Christopher J. Hogger. *Introduction to Logic Programming*. Academic Press, 1984.

[Howard 80]
W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley (editors), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[Huet 76]
Gérard Huet. *Résolution d'équations dans les langages d'ordre* 1,2,…,ω. Thèse de doctorat d'État, Université Paris VII, Paris (France), 1976.

[Huet and Plotkin 91]
Gérard Huet and Gordon D. Plotkin (editors). *Logical Frameworks*. Cambridge University Press, Cambridge (UK), 1991.

[Huntbach 86]
Matthew M. Huntbach. An improved version of Shapiro's Model Inference System. In E. Shapiro (editor), *Proceedings of ICLP'86*, pages 180–187. LNCS 225, Springer-Verlag.

[IEEE-TSE 85]
Special Issue on Artificial Intelligence and Software Engineering (edited by Jack Mostow). *IEEE Transactions on Software Engineering* 11(11), November 1985.

[Jacquet 93]
Jean-Marie Jacquet (editor). *Constructing Logic Programs*. John Wiley, 1993.

[Jantke 89]
Klaus P. Jantke. Algorithmic learning from incomplete information: Principles and problems. In J. Dassow and J. Kelemen (editors), *Machines, Languages, and Complexity*, pages 188–207. LNCS 381, Springer-Verlag, 1989.

[Jantke 91]
Klaus P. Jantke. Monotonic and non-monotonic inductive inference. *New Generation Computing* 8(4):349–360, 1991.

[Jantke and Goldammer 91]
Klaus P. Jantke and Ulf Goldammer. Inductive synthesis of rewrite rules as program synthesis. In [Clement and Lau 92], pages 65–68.

[Johansson 84]
Anna-Lena Johansson. Using symmetry for the derivation of logic programs. In S.-Å. Tärnlund (editor), *Proceedings of ICLP'*84, pages 243–251.

[Johansson 85]
Anna-Lena Johansson. *Using symmetry and substitution in program derivation*. Technical Report UPMAIL-33, Uppsala (Sweden), 1985.

[Jouannaud and Kodratoff 83]
Jean-Pierre Jouannaud and Yves Kodratoff. Program synthesis from examples of behavior. In [Biermann and Guiho 83], pages 213–250.

[Kanamori 86]
Tadashi Kanamori. *Soundness and completeness of extended execution for proving properties of Prolog programs*. Technical Report ICOT-TR-175, Tokyo (Japan), May 1986.

[Kanamori and Fujita 86]
Tadashi Kanamori and Hiroshi Fujita. Formulation of induction formulas in verification of Prolog programs. In J. H. Siekmann (editor), *Proceedings of CADE'*86, pages 281–299. LNCS 230, Springer-Verlag, 1986.

[Kanamori and Seki 86]
Tadashi Kanamori and Hirohisa Seki. Verification of Prolog programs using an extension of execution. In E. Shapiro (editor), *Proceedings of ICLP'*86, pages 475–489. LNCS 225, Springer-Verlag, 1986.

[Kant 77]
Elaine Kant. The selection of efficient implementations for a high-level language. *SIGPLAN Notices* 12(8), *SIGART Newsletter* 64, pages 140–146, August 1977.

[Kant 83]
Elaine Kant. On the efficient synthesis of efficient programs. *Artificial Intelligence* 20(3):253–305, May 1983. Also in [Rich and Waters 86a], pages 157–183.

[Kant 85]
Elaine Kant. Understanding and automating algorithm design. In [IEEE-TSE 85], pages 1361–1374. Also in *Proceedings of IJCAI'*85, pages 1243–1253.

[Kant 90]
Elaine Kant. Automated program synthesis. *University Video Communications*, Distinguished Lecture Series, Volume III, 1990.

[Kant and Barstow 81]
Elaine Kant and David R. Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering* 7(5):458–471, September 1981.

[Kodratoff and Jouannaud 84]
Yves Kodratoff and Jean-Pierre Jouannaud. Synthesizing LISP programs working on the list level of embedding. In [Biermann *et al*. 84a], pages 325–374.

[Kraan *et al*. 92]
Ina Kraan, David Basin, and Alan Bundy. Logic program synthesis via proof planning. In [Lau and Clement 93], pages 1–14.

[Lakhotia 89]
Arun Lakhotia. Incorporating "programming techniques" into Prolog programs. In E. L. Lusk and R. A. Overbeek (editors), *Proceedings of NACLP'*89, pages 426–440. MIT Press.

[Lange and Wiehagen 91]
Steffen Lange and Rolf Wiehagen. Polynomial-time inference of arbitrary pattern languages. *New Generation Computing* 8(4):361–370, 1991.

[Lassez and Marriott 87]
Jean-Louis Lassez and Kim Marriott. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning* 3:301–317, 1987.

[Lassez *et al*. 87]
Jean-Louis Lassez, M. J. Maher, and Kim Marriott. Unification revisited. In M. Boscarol, L. Carlucci Aiello, and G. Levi (editors) *Proceedings of the 1986 Workshop on the Foundations of Logic and Functional Programming*, pages 67–113. LNCS 306, Springer-Verlag, 1987.

[Lau 89]
Kung-Kiu Lau. A note on synthesis and classification of sorting algorithms. *Acta Informatica* 27:73–80, 1989.

[Lau and Prestwich 90]
Kung-Kiu Lau and S. D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In D. H. D. Warren and P. Szeredi (editors), *Proceedings of ICLP'*90, pages 667–684. MIT Press.

[Lau and Prestwich 91]
Kung-Kiu Lau and S. D. Prestwich. Synthesis of a family of recursive sorting procedures. In V. Saraswat and K. Ueda (editors), *Proceedings of SLP'*91, pages 641–658. MIT Press.

[Lau and Clement 93]
Kung-Kiu Lau and Tim Clement (editors), *Proceedings of LOPSTR'*92. Workshops in Computing Series, Springer-Verlag, 1993.

[Lavrač and Džeroski 92]
Nada Lavrač and Sašo Džeroski. Background knowledge and declarative bias in inductive concept learning. In K.P. Jantke (editor), *Proceedings of AII'*92, pages 51–71. LNCS 642, Springer-Verlag, 1992.

[Le Charlier 85]
Baudouin Le Charlier. *Réflexions sur le problème de la correction des programmes*. PhD Thesis, Facultés Universitaires Notre-Dame de la Paix, Namur (Belgium), 1985.

[Lichtenstein and Shapiro 88]
Yossi Lichtenstein and Ehud Shapiro. Abstract algorithmic debugging. In R. A. Kowalski and K. A. Bowen (editors), *Proceedings of ICLP'*88, pages 512–531. MIT Press.

[Lloyd 87]
John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second, extended edition, 1987.

[London and Feather 82]
Philip E. London and Martin S. Feather. Implementing specification freedoms. *Science of Computer Programming* 2:91–131, 1982. North-Holland. Also in [Rich and Waters 86a], pages 285–305.

[Lowry and Duran 89]
Michael Lowry and Raul Duran. Knowledge-based software engineering. In A. Barr, P. R. Cohen, and E. A. Feigenbaum (editors), *The Handbook of Artificial Intelligence* (Volume IV), pages 241–322. Addison Wesley, 1989.

[Manna 74]
Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[Manna and Waldinger 77]
Zohar Manna and Richard Waldinger. The automatic synthesis of systems of recursive programs. In *Proceedings of IJCAI'77*, pages 405–411. Early version of [Manna and Waldinger 79].

[Manna and Waldinger 79]
Zohar Manna and Richard Waldinger. Synthesis: Dreams ⇒ Programs. *IEEE Transactions on Software Engineering* 5(4):294–328, July 1979.

[Manna and Waldinger 80]
Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems* 2(1):90–121, January 1980. Also in [Webber and Nilsson 81], pages 141–172. Also in [Biermann *et al*. 84a], pages 33–68. Also in [Rich and Waters 86a], pages 3–34.

[Manna and Waldinger 81]
Zohar Manna and Richard Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming* 1:5–48, 1981. North-Holland. Also in [Biermann and Guiho 83], pages 251–307.

[Manna and Waldinger 87]
Zohar Manna and Richard Waldinger. The origin of a binary-search paradigm. *Science of Computer Programming* 9:37–83, 1987. North-Holland.

[Manna and Waldinger 91]
Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. In F. L. Bauer (editor), *Logic, Algebra, and Computation*, pages 41–107. Volume ASI-F79, Springer-Verlag, 1991.

[Martin-Löf 79]
Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the* 6^{th} *International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, 1979. Published by North Holland, 1982.

[McCune 77]
Brian McCune. The PSI program model builder: Synthesis of very high-level programs. *SIGPLAN Notices* 12(8), *SIGART Newsletter* 64, pages 130–139, August 1977.

[Michalski 84]
Ryszard S. Michalski. Inductive learning as rule-guided generalization of symbolic descriptions: A theory and implementation. In [Biermann *et al*. 84a], pages 517–552.

[Mitchell 81]
Tom M. Mitchell. Generalization as search. In [Webber and Nilsson 81], pages 517–542. Also in *Artificial Intelligence* 18(2):203–226, March 1982.

[Mitchell *et al*. 86]
Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning* 1(1):47–80, 1986.

[Mohring 86]
Christine Mohring. Algorithm development in the calculus of constructions. In *Proceedings of the* 1986 *IEEE Symposium on Logic in Computer Science*, pages 84–95.

[Muggleton 91]
Stephen Muggleton. Inductive logic programming. *New Generation Computing* 8(4):295–317, 1991. Also in [Muggleton 92], pages 3–27.

[Muggleton 92]
Stephen Muggleton (editor). *Inductive Logic Programming*. Volume APIC-38, Academic Press, 1992.

[Muggleton 94]
Stephen Muggleton. *Foundations of Inductive Logic Programming*. Oxford University Press, 1994. (Forthcoming.)

[Muggleton and Buntine 88]
Stephen Muggleton and Wray Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the* 1988 *International Conference on Machine Learning*, pages 339–352. Morgan Kaufmann, 1988. Also in [Muggleton 92], pages 261–280.

[Muggleton and Feng 90]
Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Proceedings of ALT'*90. Ohmsha (Tokyo, Japan), 1990. Also in [Muggleton 92], pages 281–298.

[Neugebauer 92]
Gerd Neugebauer. The LOPS approach: A transformational point of view. In [Lau and Clement 93].

[Nordström *et al*. 90]
Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. The International Series of Monographs in Computer Science, Volume 7, Clarendon Press, Oxford (UK), 1990.

[O'Keefe 90]
Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[Partsch and Steinbrüggen 83]
Helmut Partsch and R. Steinbrüggen. Program transformation systems. *Computing Surveys* 15(3):199–236, September 1983.

[Pettorossi and Proietti 89]
Alberto Pettorossi and Maurizio Proietti. Decidability results and characterization of strategies for the development of logic programs. In G. Levi and M. Martelli (editors), *Proceedings of ICLP'*89, pages 539–553. MIT Press.

[Pitt and Valiant 88]
Leonard Pitt and Leslie G. Valiant. Computational limits on learning from examples. *Journal of the ACM* 35(4):965–984, October 1988.

[Plotkin 70]
Gordon D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie (editors). *Machine Intelligence* 5:153–163, 1970. Edinburgh University Press, Edinburgh (UK).

[Plotkin 71]
Gordon D. Plotkin. A further note on inductive generalization. In *Machine Intelligence* 6:101–124, 1971. Edinburgh University Press, Edinburgh (UK).

[Popplestone 70]
R. J. Popplestone. An experiment in automatic induction. In B. Meltzer and D. Michie (editors). *Machine Intelligence* 5:203–215, 1970. Edinburgh University Press, Edinburgh (UK).

[Quinlan 90]
J. R. Quinlan. Learning logical definitions from relations. *Machine Learning* 5(3):239–266, August 1990.

[Reynolds 70]
John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence* 5:135–151, 1970. Edinburgh University Press, Edinburgh (UK).

[Rich 81]
Charles Rich. A formal representation for plans in the Programmer's Apprentice. In *Proceedings of IJCAI'*81, pages 1044–1052. Also in [Rich and Waters 86a], pages 491–506.

[Rich and Waters 86a]
Charles Rich and Richard C. Waters (editors). *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann, 1986.

[Rich and Waters 86b]
Charles Rich and Richard C. Waters. Introduction. In [Rich and Waters 86a], pages *xi–xxi*.

[Rich and Waters 87]
Charles Rich and Richard C. Waters. *The Programmer's Apprentice: A program design scenario*. AI Memo 933A, Massachusetts Institute of Technology, Cambridge (MA, USA), 1987.

[Rich and Waters 88a]
Charles Rich and Richard C. Waters. Automatic programming: Myths and prospects. *IEEE Computer* 21(8):40–51, August 1988.

[Rich and Waters 88b]
Charles Rich and Richard C. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer* 21(11):10–25, November 1988.

[Rich and Waters 89]
Charles Rich and Richard C. Waters. *Intelligent assistance for program recognition, design, optimization, and debugging*. AI Memo 1100, Massachusetts Institute of Technology, Cambridge (MA, USA), 1989.

[Rouveirol 91]
Céline Rouveirol. Semantic model for induction of first order theories. In *Proceedings of IJCAI'*91, pages 685–690.

[Ruth 78]
Gregory R. Ruth. Protosystem I: An automatic programming system prototype. In *Proceedings of the AFIPS'*78 *Conference*. Also in [Rich and Waters 86a], pages 215–221.

[Sammut and Banerji 86]
Claude Sammut and Ranan B. Banerji. Learning concepts by asking questions. In R. Michalski, J. G. Carbonell, and T. Mitchell (editors), *Machine Learning: An Artificial Intelligence Approach*, Volume 2, pages 167–192. Morgan Kaufmann, 1986.

[Sato and Tamaki 84]
Taisuke Sato and Hisao Tamaki. Transformational logic program synthesis. In *Proceedings of the International Conference on Fifth-Generation Computer Systems*, pages 195–201, 1984.

[Sato and Tamaki 89]
Taisuke Sato and Hisao Tamaki. First-order compiler: A deterministic logic program synthesis algorithm. *Journal of Symbolic Computation* 8(6):605–627, 1989.

[Shapiro 81]
Ehud Y. Shapiro. An algorithm that infers theories from facts. In *Proceedings of IJCAI'*81, pages 446–451.

[Shapiro 82]
Ehud Y. Shapiro. *Algorithmic Program Debugging*. PhD Thesis, Yale University, New Haven (CT, USA), 1982. Published under the same title by MIT Press, 1983.

[Shaw *et al*. 75]
David E. Shaw, William R. Swartout, and Cordell Green. Inferring LISP programs from examples. In *Proceedings of IJCAI'*75, pages 260–267.

[Siklóssy and Sykes 75]
L. Siklóssy and D. A. Sykes. Automatic program synthesis from example problems. In *Proceedings of IJCAI'*75, pages 268–273.

[Sintzoff 93]
Michel Sintzoff. Private communication. May 1993.

[Smith 81]
Douglas R. Smith. A design for an automatic programming system. In *Proceedings of IJCAI'*81, pages 1027–1029.

[Smith 82]
Douglas R. Smith. Derived preconditions and their use in program synthesis. In D. W. Loveland (editor), *Proceedings of CADE'*82, pages 172–193.

[Smith 84]
Douglas R. Smith. The synthesis of LISP programs from examples: A survey. In [Biermann *et al*. 84a], pages 307–324.

[Smith 85]
Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27(1):43–96, 1985. Also in [Rich and Waters 86a], pages35–61.

[Smith 88]
Douglas R. Smith. *The structure and design of global search algorithms*. Technical Report KES.U.87.12, Kestrel Institute, Palo Alto (CA, USA), 1988.

[Smith 90]
Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering* 16(9):1024–1043, September 1990.

[Smith *et al*. 85]
Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold. Research on knowledge-based software environments at Kestrel Institute. In [IEEE-TSE 85], pages 1278–1295.

[Soloway and Ehrlich 84]
Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering* 10(5):595–609, September 1984. Also in [Rich and Waters 86a], pages 507–521.

[Steier and Kant 85]
David M. Steier and Elaine Kant. The roles of execution and analysis in algorithm design. In [IEEE-TSE 85], pages 1375–1386.

[Steier and Anderson 89]
David M. Steier and A. P. Anderson. *Algorithm Synthesis: A Comparative Study*. Springer-Verlag, 1989.

[Sterling and Shapiro 86]
Leon S. Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 1986.

[Sterling and Kirschenbaum 91]
Leon S. Sterling and Marc Kirschenbaum. Applying techniques to skeletons. In [Jacquet 93], pages 127–140.

[Summers 77]
Phillip D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM* 24(1):161–175, January 1977. Also in [Rich and Waters 86a], pages 309–316.

[Takayama 87]
Yukihide Takayama. Writing programs as QJ proof and compiling into Prolog programs. In *Proceedings of SLP'*87, pages 278–287.

[Tamaki and Sato 84]
Hisao Tamaki and Taisuke Sato. Unfold/fold transformations of logic programs. In S.-Å. Tärnlund (editor), *Proceedings of ICLP'*84, pages 127–138.

[Tärnlund 78]
Sten-Åke Tärnlund. An axiomatic data base theory. In H. Gallaire and J. Minker (editors), *Logic and Databases*, pages 259–289. Plenum Press, 1978.

[Tärnlund 81]
Sten-Åke Tärnlund. *A programming language based on a natural deduction system*. Technical Report UPMAIL-6, Uppsala (Sweden), 1981.

[Tinkham 90]
Nancy L. Tinkham. *Induction of Schemata for Program Synthesis*. PhD Thesis, Duke University, Durham (NC, USA), 1990.

[Valiant 84]
Leslie G. Valiant. A theory of the learnable. *Communications of the ACM* 27(11):1134–1142, November 1984.

[Valiant 85]
Leslie G. Valiant. Learning disjunctions of conjunctions. In *Proceedings of IJCAI'85*, Volume 1, pages 560–566.

[van Lamsweerde 91]
Axel van Lamsweerde. Learning machine learning. In A. Thayse (editor), *From Natural Language Processing to Logic for Expert Systems*, pages 263–356. John Wiley, 1991.

[Vere 75]
Steven A. Vere. Induction of concepts in the predicate calculus. In *Proceedings of IJCAI'*75, pages 281–287.

[Waldau 91]
Mattias Waldau. Formal validation of transformation schemata. In [Clement and Lau 92], pages 97–110.

[Waldinger and Lee 69]
Richard J. Waldinger and Richard C. T. Lee. PROW: A step toward automatic program writing. In *Proceedings of IJCAI'*69, pages 241–252.

[Waterman *et al*. 84]
D. A. Waterman, W. S. Faught, Philip Klahr, Stanley J. Rosenschein, and Robert Wesson. Design issues for exemplary programming. In [Biermann *et al*. 84a], pages 433–460.

[Waters 85]
Richard C. Waters. The Programmer's Apprentice: A session with KBEmacs. In [IEEE-TSE 85], pages 1296–1320. Also in [Rich and Waters 86a], pages 351–375.

[Webber and Nilsson 81]
Bonnie Lynn Webber and Nils J. Nilsson (editors). *Readings in Artificial Intelligence*. Morgan Kaufmann, 1981.

[Wiggins 92]
Geraint Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. Apt (editor), *Proceedings of JICSLP'*92, pages 351–365.

[Wiggins *et al*. 91]
Geraint Wiggins, Alan Bundy, Ina Kraan, and Jane Hesketh. Synthesis and transformation of logic programs from constructive, inductive proof. In [Clement and Lau 92], pages 27–45.

[Wile 83]
David S. Wile. Program developments: Formal explanations of implementations. *Communications of the ACM* 26(11):902–911, 1983. Also in [Rich and Waters 86a], pages 191–200.

[Zloof 77]
M. Zloof. Query by example: A data base language. *IBM Systems Journal* 4(16):324–343, 1977.