Iterative Induction of Logic Programs

An approach to logic program synthesis from incomplete specifications

Alípio Mário Guedes Jorge

Tese submetida para obtenção
do grau de Doutor em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

January 1998

To my Parents.

To my Wife.

Acknowledgements

My supervisor, Pavel Brazdil, has been an unlimited source of encouragement, enthusiasm and patience. Thanks for his valuable comments, suggestions and guidance.

My colleagues in LIACC provided an excellent working atmosphere, with relaxing breaks whenever necessary. Special thanks to those in NIAAD (the machine learning group), particularly to the ones who share the office with me: João Gama and Luís Torgo.

I must thank the Portuguese agency JNICT (Programa Ciência, grant BD/1327/91/IA and PRAXIS XXI, grant BD/3285/94) for the financial support without which this work would have not been possible. I also thank MLNet, ILPNet, Faculdade de Economia da U. Porto, and the Japanese agency IISF, which made possible my participation in several international scientific events.

Thanks to all my friends (I fortunately have a good collection of them) for all the good moments, dinners, parties, weekends, etc. Special thanks to those more directly involved with my work: Mário Florido and Paulo Azevedo.

Thanks to my family, for their support and care (I have a great family too).

Special thanks to my wife, Xinha, for her love and companion, and to my baby-daughter Carolina for giving me peaceful nights since she was 3 months old.

Thanks to my parents, for their love, support and encouragement ever since.

Contents

1. Introduction

In this thesis we describe a methodology for the automatic construction of Prolog programs from various pieces of available information. Programs are described in terms of positive and negative examples, sketches and integrity constraints. Definitions of auxiliary predicates and knowledge about the structure of the clauses to construct are also given. This methodology is implemented as system SKIL (Sketch-based Inductive Learner) and its iterative extension SKILit. Both systems are written in Prolog.

The information given to the system describes how the intended program should behave and can be regarded as a program specification. Since we are dealing with fragmented information we have an *incomplete specification* which does not fully describe the behaviour of the program. The unspecified behaviour is hypothesized by our methodology by means of inductive inference. For that reason, we can see our work as an approach to the *inductive synthesis of logic programs from incomplete specifications*. This work is therefore related to the more general field of *Automatic Programming* or *(Automatic) Program Synthesis*.

On the other hand, inductive synthesis of logic programs can be naturally regarded as a sub-field of *Inductive Logic Programming* (ILP). The aim of ILP is to induce theories

from observations using logic programming formalisms to describe both theories and observations. For that reason it is usually regarded as an intersection of *Machine Learning* and *Logic Programming* (Figure 1.1).

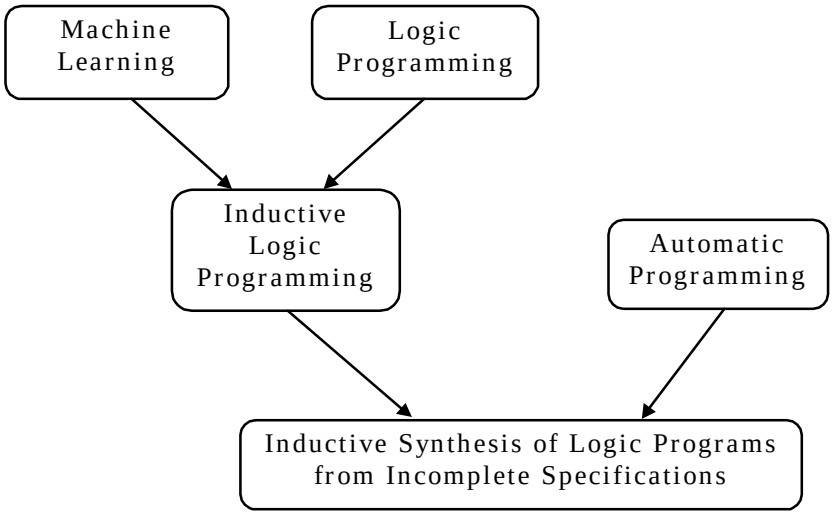

Figure 1.1: Our work and related fields.

Let us take a look at a simple example of what we mean by inductive synthesis of logic programs from incomplete specifications. Given the set E^+ of positive examples of the relation *descendant/2*

descendant(alipio,antonio).
descendant(alipio,adriana).

and the set of negative examples E^- of the same relation

descendant(antonio,alipio).
descendant(adriana,antonio).

and an auxiliary program *B* (which is often referred to as background knowledge)

son(antonio,adriana).
son(alipio,antonio).

a logic program *P* defining predicate *descendant/2* is constructed:

descendant(A,B)←son(A,B).

descendant(A,B)←son(A,C), descendant(C,B).

In terms of program synthesis, the specification is made of the example sets E^+ and E^-. Program P is a synthesized program which, together with the auxiliary program B, satisfies the specification. In terms of ILP, examples E^+ and E^- are regarded as observations. These are explained by the induced theory P together with background knowledge B. The examples in E^+ are logical consequences of $P \cup B$ whereas the ones in E^- are not.

The conditions under which P satisfies the incomplete specification $\{E^+,E^-\}$, or P explains the observations $\{E^+,E^-\}$ with respect to B can be stated as:

$$P \cup B \models E^+ \qquad \text{and} \qquad P \cup B \not\models e^- \text{ for all } e^- \in E^-$$

The general aim of an ILP system, whether or not regarded as a program synthesis system, is to find a program P which satisfies the above conditions. This thesis describes the methodology behind one such system: SKILit.

1.1 Motivation

One of the main motivations of this work was the fact that many ILP techniques and algorithms did not seem to be well suited to the problem of inductive program synthesis, and in particular to the synthesis of recursive programs. ILP systems which represented the state-of-the-art when this work first started, such as FOIL and GOLEM, were practically unable to handle incomplete sets of examples. In order to construct the definition of a recursive predicate, such systems require large numbers of well chosen examples. The system SKILit we propose is able to induce recursive definitions from small sparse sets of examples. Experiments show that SKILit obtains good results when only few positive examples are available even if they are randomly generated. This is due

to the iterative induction technique employed by SKILit, which is one of the main contributions of the present work.

Other more recent systems also have this ability to induce recursive clauses from a sparse set of positive examples. However, these other systems have a strong language bias and can only synthesize programs within a restricted family of programs. Using the methodology described in this thesis, system SKILit is potentially able to induce any pure Prolog program since it allows the declaration of programming knowledge through clause structure grammars. These are represented using the definite clause grammar notation (DCG). We should stress, however, that SKILit is able to perform synthesis when no grammar is provided.

Another problem we approach in this thesis is related to the large number of negative examples required by most systems to avoid the induction of over-general programs. Our methodology enables the use of integrity constraints to express the bounds of the intended relation. The use of integrity constraints in ILP is not new. However, processing such constraints usually involves heavy theorem proving mechanisms. The approach we adopt here for integrity constraint checking is a very efficient one. It is based on a Monte Carlo strategy which, given an integrity constraint *I* and an induced program *P*, checks with some degree of uncertainty whether *P* and *I* are consistent or not.

1.2 Main contributions

The methodology presented in this thesis combines some novel techniques with existing methods. Our main contributions are SKIL's inductive engine, iterative induction, and an efficient Monte Carlo method to handle integrity constraints. The basic inductive engine presented is adequate for program synthesis from few examples. It also exploits mode and type information, as well as programming knowledge represented as clause structure grammars and algorithm sketches. Algorithm sketches allow the user to represent

specific programming knowledge and give this information to the system. Iterative induction allows more flexibility in the choice of the positive examples given to a system. The Monte Carlo constraint handler makes it practical to use integrity constraints in inductive program synthesis. A brief overview of each one of these aspects is given in the following sections.

1.2.1 The inductive engine

From a specification including the positive examples

member(2,[2]).
member(2,[1,2]).

The inductive engine of SKIL is able to induce the clauses

member(A,[A|B]). (C1)
member(A,[B|C])←member(A,C). (C2)

Our methodology constructs each clause by searching for a relational link from the input to the output arguments of some positive example. The connection is established using the auxiliary predicates defined in the background knowledge and the positive examples initially given. The input/output modes declared for each predicate are also taken into account. For example, assuming that the second argument is output and the first one is input, the arguments of *member(2,[1,2])* can be relationally linked as follows. From *[1,2]* we get terms *1* and *[2]* by decomposing the list *[1,2]* and from *[2]* we get term *2*, using the example *member(2,[2])*. This link corresponds to the following instance of clause (C2):

member(2,[1,2])←member(2,[2]).

This instance is turned into a clause by replacing terms with variables.

The search for a relational link is guided by an example (data-driven induction), which has the advantage of reducing the number of candidate clauses to consider. The strategy

for constructing each clause depends on one positive example only at a time. The reason for this is that our inductive engine does not employ heuristics based on example coverage or similar notions, as it happens with FOIL [96] or CHILLIN [125]. These heuristics tend to be less reliable when few examples are available.

Our inductive engine also exploits programming knowledge represented as clause structure grammars. This is a very simple and powerful formalism which can be seen also as declarative bias.

The inductive engine also allows the synthesis from algorithm sketches. These can be seen as partially explained positive examples which speed-up the synthesis process.

For example, the positive example *member(6,[3,1,6,5])* could be partially explained by telling the system that from list *[3,1,6,5]* you obtain list *[1,6,5]* and from this list you obtain *6*, the desired output. This information can be represented as an algorithm sketch and be given to the system. The sketch is represented as a ground clause.

member(6,[3,1,6,5])←$P1([3,1,6,5], [1,6,5]), $P2([1,6,5],6).

The *$P1* and *$P2* predicates represent unknown sequences of literals involving operational predicates. The synthesis task consists mainly of constructing those sequences of literals. Any positive example like *member(2,[1,2])* can be represented by a sketch like *member(2,[1,2])←$P3(2,[1,2])*.

Our inductive engine handles both plain positive examples and algorithm sketches in a uniform way. Each clause is obtained from one example or sketch by using a unique sketch refinement operator. This sketch refinement operator is shown to be complete under adequate assumptions.

1.2.2 Iterative induction

In order to induce the recursive clause from example *member(2,[1,2])*, the inductive engine of SKIL needs to be given the example *member(2,[2])*. This fact makes the

induction of recursive programs by SKIL difficult when examples are not carefully chosen. The role of iterative induction is to facilitate the synthesis of recursive programs. System SKILit implements iterative induction.

Suppose that the specification now includes the positive examples

member(7,[7,9]).
member(2,[1,2]).

From this specification SKILit is able to synthesize the same recursive definition we saw in the previous Section.

member(A,[A|B]). (C1)
member(A,[B|C])←member(A,C). (C2)

Let us see, in broad terms, how. In the first iteration two clauses are constructed, one for each positive example.

member(A,[A|B]).
member(A,[B,A|C]). (C3)

In the second iteration, positive examples are again processed. The recursive clause C2 is constructed from the example *member(2,[1,2])* with the help of the fact *member(2,[2])*. However, this fact is not in the specification. It is instead covered by clause C1. This clause has a very important role in the inductive process.

The clauses induced during the first iterations are used by the system to support the introduction of recursive clauses. They express certain properties of the relation to synthesize. These properties may or may not be part of the final program. The properties made redundant by other clauses are deleted by SKILit's program compression module, TC.

1.2.3 Integrity constraints and the Monte Carlo method

The module MONIC of system SKILit processes integrity constraints by using a rather efficient, although incomplete, Monte Carlo strategy. Every program *P* synthesized by SKILit should satisfy the integrity constraints in the specification. Satisfaction checking is done by randomly generating *n* facts which are logical consequences of the program. Each one of these facts is used to look for a violating instance of some integrity constraint.

For instance, the integrity constraint for predicate *union/3*

union(A,B,C),member(X,C)→member(X,A),member(X,B)

is read as: "if *X* is in list *C*, then it is either in *A* or in *B*". This constraint must be respected by the program that defines the predicate *union/3*. Given a candidate program *P* with *union([2],[],[3])* as a logical consequence, and a correct definition for the predicate *member/2*, one violating instance of the above integrity constraint is

union([2],[],[3]),member(3,[3])→member(3,[]),member(3,[2])

since the antecedent is true and the consequent is false.

Our constraint checker MONIC does not necessarily find a violating instance of the integrity constraint. This only happens if one of the *n* randomly drawn logical consequences of *P* results in a violating instance as shown above. The probability of that to happen grows with *n*, which can be set by the user.

1.3 Overview of the thesis

In Chapter 2, we situate the current work in the context of program development. We refer to CASE tools, formal methods, deductive synthesis and inductive synthesis. In Chapter 3, we discuss Inductive Logic Programming (ILP). We start with an

introduction to Logic Programming, and present the ILP concepts and techniques which are relevant to our work.

In Chapter 4, we present the inductive engine that is the core of our methodology. It is described as system SKIL, which synthesizes logic programs by exploiting examples and sketches. We give a sketch refinement operator and show a completeness result for it. In Chapter 5, we introduce the iterative induction technique that overcomes the main limitation of SKIL: the difficulty of inducing recursive definitions from sparse sets of positive examples. System SKILit (SKIL iterative version) iteratively invokes (sub-) system SKIL. In Chapter 6, we provide empirical evaluation of the method and of system SKILit. In Chapter 7, we describe the constraint checker MONIC, which uses a Monte Carlo strategy. MONIC allows the inclusion of integrity constraints in the specifications given to SKILit. In Chapter 8, we give conclusions, limitations and future work.

2. Program Development

In this chapter we give a brief overview of various methodologies of program development, including software engineering and automatic programming, covering CASE tools and formal program development. A greater attention is given to the synthesis of programs from incomplete specifications, particularly to the synthesis of logic programs from examples.

2.1 Introduction

Software engineering traditionally divides program development in four distinct phases [113].

- *Elaboration of the specification*. The specification contains the user's requirements relative to the program to construct. The requirements are described in natural language. The *specification* should contain information about what the program should do without describing how it should be done.

- *Analysis and design*, which elaborates the items given in the specification. In this phase program developers make a high level description of the involved algorithms. Data structures and data-flow are identified.

- *Implementation*, where the high level algorithms designed in the previous phase are translated into executable code[1].

- *Verification*, where the executable program is confronted with the specification. If any deficiency is found in the program (i.e. the program is incorrect), one or more of the previous phases are redone.

Our work intends to contribute to the automation of code generation within the scope of programming in the small[2], whilst permitting incomplete specifications by examples, and other pieces of information. On the one hand, it is our aim to make specifications as simple to construct as possible, on the other we wish to totally or partially automate the generation of code from incomplete specifications.

2.2 Automatic programming

Could the computer accomplish the laborious task of programming? This dream is as old as programming itself. The quest for automatic programming is motivated by two main reasons:

- to accelerate the process of program development, mainly the implementation phase previously referred, freeing as much as possible the analyst/programmer from non-creative tasks;

---

[1] The expression 'executable code' is used in the sense that there is an available interpreter/compiler for that language.

2 A distinction is also made between *programming in the large* and *programming in the small*. Programming in the large involves a large team of analysts and programmers, working for a long period of time (months to years),

- to increase the reliability of programs, minimizing human intervention, which is often a source of errors.

Computer aiding tools for software development, the formal development methodologies, and the synthesis of programs have pursued these objectives.

2.3 CASE tools

The acronym *CASE* stands for *Computer Aided Software Engineering*. *CASE Tools* are computer programs that aid the task of developing a system, from the elaboration of specifications to the production of documentation [113].

A CASE system can contain several different tools. Diagram editors for the management of application related information: data flow, system structure, entities-relationships diagrams, etc. These editors are usually more than simple design tools. They should be able to capture the information contained in the diagrams and alert the user for inconsistencies and other anomalies. Other sorts of CASE tools include database querying tools, dictionaries which maintain the information relative to the involved entities, tools which allow easy generation of reports, user interface generators, etc.

CASE tools enable greater productivity in the development of complex systems, and are common in professional environments nowadays. Its main role is to organize the vast quantity of information involved in a large development project, in order to make that information easily accessible to everyone involved. Systems developed with the support of CASE tools tend to be more reliable.

---

while programming in the small refers to systems which do not take more than a few months to develop with no more than one or two people. ‘Software’ engineering is especially devoted to programming in the large.

Some CASE systems include *code generators*. These are able to create preliminary segments of code (*skeleton code*) from the information gathered in the diagram editors and data dictionaries. Even so, the availability of this type of CASE tool is very limited.

To conclude, CASE tools are mainly useful for the support of the management of project development. Most tedious programming tasks are still left to the programmer. Yet, without tools capable of automating or semi-automating the generation of code, the CASE technology is far from reaching its full potential [35].

2.4 Formal methods

In terms of formal development methodologies, programming is seen as a mathematical activity, and programs are considered complex mathematical expressions [45]. This conception of programming allows, for example, to prove that a program is correct with respect to its specification [30].

In approaches based on formal methods, the specification is expressed in a formal language, as first order logic [28], instead of natural language. The executable program can be obtained from the given specification using inference and/or rewrite rules. The application of these rules can be manual or semi-automatic. In general, it is difficult to mechanically derive a complex program this way [28]. For this reason, we frequently find program synthesis methodologies which are semi-automatic and guided by the user. We give two examples below.

The KIDS system by Douglas Smith, supports the development of correct and efficient programs from formal specifications (cf. following Section). The environment for the development of KIDS is highly automated, although interactive. The user makes high level decisions concerning program the design and the system takes these decisions into account generating an executable program [111].

Jüllig [55] proposes a program development environment (REACTO) where the spirit of CASE tools is integrated with formal methods, and with program synthesis. On the one hand, graphic aiding tools for analysis are made available to the user, on the other hand the user is allowed to write formal specifications and obtain executable code. One of the components of REACTO is the KIDS system previously referred to.

In conclusion, CASE tools provide graphical aid for analysis, but give limited support for the generation of code. Formal methods are mostly used for writing the specifications rather than for the generation of an executable program. Program synthesizers, discussed in the next Section, concentrate on the generation of code instead of system analysis [55].

2.5 Program synthesis

Broadly speaking, we call *program synthesis* to any systematic process of program construction from a given specification which describes what the program should do [27]. Within the category of systematic methods we find the (semi-)automatic methods of code generation from a specification of the intended program behaviour. In this case, the program synthesis is also named as *automatic programming* [8,99].

In this context, the term 'specification' can have many different connotations. Biermann organises the automatic programming research field according to the kind of specification used [8]: synthesis from formal specifications (first order logic formulas); synthesis from examples of input/output pairs; and synthesis from dialogues in natural language between the synthesis system and the user. We can also find formal specifications represented as a hierarchical finite state machine [55] or a temporal logic [108].

When the specification is expressed in natural language, the code generator must cope with the typical ambiguity and syntactical irregularity of natural language. Synthesis

systems from natural language are usually interactive, allowing the user to describe the problem through a dialogue with the system.

In the 1970´s there were a few ambitious projects in this domain [42] with limited success. Later, the research focus moved in the direction of specifications in *very high level languages*. These languages are closely related to formal languages, even though they sometimes allow some informality typical of natural language [99,part v].

2.5.1 Logic program synthesis

Within program synthesis, we are mainly interested in *logic program synthesis*. In this field, Deville and Lau [27] divide the formal specifications into *formal* and *informal* ones. The formal specifications can be either *complete* or *incomplete*.

A formal specification is expressed using a formal language, as the first order logic or one of its subsets. The specification is a set of logical formulas involving one logical predicate *r* which is to be defined. This notion of specification in the context of logic program synthesis is broad enough to include complete and incomplete specifications.

A *complete specification* includes all the conditions which the program to synthesize should satisfy. An *incomplete specification* describes only part of those conditions. In general, a specification from examples of answers of a logic program is incomplete, i.e., not all of the program behaviour is specified. In this case, the code generator will have the task of hypothesizing the unspecified behaviour. A specification from examples can be regarded as a formal specification, as long as a rigorous language is used to describe the examples [27].

Example 2.1: (from [27]) Two specifications for the predicate *included(X,Y)* which defines the set of pairs <*X*,*Y*>, of which *X* and *Y* are lists and every element of *X* is contained in *Y*.

Complete specification:

$\{included(X,Y) \leftrightarrow \forall A(\ member(A,X) \rightarrow member(A,Y)\ )\ \}$

Incomplete specification (by examples):

$\{\ included([],[2,1]),\ included([1,2],[1,3,2,1]),\ \neg included([2,1],[])\ \}$

♦

Logic program synthesis from formal specifications has three main approaches:

- *constructive synthesis*, whereby a program is extracted from a constructive proof of the existence of a program satisfying the specification;
- *deductive synthesis*, whereby a program is derived from a specification using deduction rules;
- *inductive synthesis*, whereby a program which generalizes the information contained in the specification is constructed using inductive methods.

Among these three approaches to program synthesis from formal specifications, our work can be regarded as inductive synthesis, more specifically, as *inductive synthesis from examples* of the intended program behaviour.

2.5.2 Program synthesis from examples

In the early 1980´s, the MIS system (*Model Inference System*) by Ehud Shapiro [109] synthesizes programs in Prolog language from examples given by the user.

System MIS works interactively following a *program debugging* philosophy. The user presents positive and negative examples and the system confronts the given examples with the current version of the program, starting with the empty program. When a new example highlights a problem in the program, the system modifies it with the aim to eliminate the error.

Debugging a program consists of the elimination, creation or modification of individual clauses. During the debugging process, the system may query the user about predicates involved in the program. These queries have the form *"is p(a) true or false?"* (membership queries or ground queries) and *"which values of the variable X make p(X) true?"* (existential queries).

The system MIS is able to generate small Prolog programs, such as *member/2*, which is true if the first argument is a member of the second argument, or *append/3*, which concatenates two lists into a third one. The system can also be adapted to generate programs in DCG (*definite clause grammar*) notation.

The work of Ehud Shapiro contains a methodology for the synthesis of Prolog programs from examples which still inspires work on the subject [41,95]. The influence of his work is mainly noticeable in the field of *Inductive Logic Programming* (ILP, cf. Chapter 3).

ILP came about in the nineties and its main concern is to generate logic programs from examples [77]. Although the main focus of ILP research has not been automatic programming many ILP systems demonstrate their abilities by showing that it is possible to generate simple Prolog programs at the level of the ones taught in a first logic programming course. As examples of some approaches concerned with automatic programming we can refer to the works of Quinlan [96,97], Bergadano et al.[5, 6], Flener [37,39] and Popelinsky et al. [94].

Although currently the trend is to do inductive synthesis with logic programming languages such as Prolog, in the seventies and eighties the preferred language was LISP, which is closer to the functional paradigm. The works of Summers [118] and Biermann [7] are examples of that.

The shift from functional to logic languages, may be attributed to the appropriateness of logic programming to the task of inducing clauses from examples, and also to the growing popularity of Prolog. The fact that in a logic programming language

generalization and specialization of clauses and programs correspond to very simple operations[3] may have contributed to that shift [109]. Despite the general trend, work on inductive synthesis of functional programs is still published. In 1995 system ADATE [89] synthesizes programs in the language ML.

To sum up, we can say that both logic programming and functional languages have important features which justify their choice for program synthesis (and not from examples only). Both paradigms have meta-programming capabilities, which are important for automatic programming. Both LISP and Prolog programs tend to be compact and relatively easy to understand. Finally, both functional and logic languages have strong theoretical foundations, which enables a clear formalization of inductive operations and program transformation [109, pp.162-163].

2.6 Other relevant topics

Other subjects in computer science are relevant to the quest for computer tools that ease the effort of programmers and program analysts. We will not refer to these subjects in a systematic way, but rather present some pointers which can be followed.

- *Program development* environments. In the area of Logic Programming we highlight the work of Mireille Ducassé [31]. A good example of how a program development environment can help a FORTRAN programmer to exploit the existing sub-routine library can be found in the work of Stickel et al. [117].

- *Algorithmic debugging*. This is an important source of inspiration for the work on synthesis from examples. The synthesis process can be seen as a debugging process starting with the empty program. In this area we have, among others, the works of Shapiro [109], Moniz Pereira and Miguel Calejo [17,91] and Paaki et al.[90].

---

[3] These operations are simple when the generalization model employed is θ-subsumption. Other generalization

- *Programming by demonstration*. The aim of programming by demonstration is, according to Cypher, the following: If a user knows how to accomplish a task using a computer, that should be enough to create a program that automates that task. It should not be necessary to know a programming language like C or BASIC. Typically, the user demonstrates his/her intentions by means of a graphical interface. The system based on these notions generalizes the user's actions and infers a program or a macro [19].

2.7 Summary

CASE tools provide a good help for the tasks of project analysis and development, increasing the overall productivity of software development. They also provide greater software reliability. However, CASE methodology has offered very little regarding code generation, leaving many tedious tasks to the programmer.

The advocates of formal development methods regard programming as a mathematical activity and programs as complex mathematical expressions [45]. They propose formal specification languages, from which one obtains the program code following a formal methodology. The correctness of a formally developed program with respect to its specification can be proved. Employing such rigorous approach helps to avoid many programming errors. This aspect is especially relevant in critical applications such as air traffic control or industrial plant maintenance, where a program bug may have dramatic costs [30].

The systematic development of programs from specifications is referred here as program synthesis. Specifications can be formal or informal, complete or incomplete. From formal and complete specifications, programs can be derived using deductive methods similar to the ones used in theorem proving..

---

models may be more complex.

Formal methods, however, are often too heavy. Formal programming is only within the reach of experts. To write a formal and complete specification is not an easy task. Existing derivation methods are not totally automated, and still demand much effort from the programmer. Program synthesis from incomplete specifications (inductive synthesis) facilitates the task of the programmer by eliminating the need for abstraction demanded by traditional formal methods.

The area of inductive synthesis is usually geared towards the generation of LISP and Prolog programs from examples. While the synthesis of LISP programs from examples has seen little development in the nineties, the synthesis of Prolog programs has greatly increased with the growth of the fields of logic programming and of inductive logic programming (ILP).

ILP technology may be an important component of future programming environments. The aim of having one day a totally automated tool which constructs any intended program from examples only (Biermann calls that *auto-magic programming*) seems unrealistic. However, we believe ILP can give an important contribution to the development of tools that help the programmer accomplishing his task. Moreover, ILP may enable unskilled computer users to create small computer programs without programming, and therefore increase dramatically the ease of constructing new applications.

3. Inductive Logic Programming

In this chapter we introduce Inductive Logic Programming (ILP) concepts which are relevant to our work. We start with Logic Programming itself, which can be seen as one of the pillars of ILP. The general ILP task is defined as the construction of a logic program satisfying certain conditions. The main ILP approaches are described. We conclude the Chapter by giving an account of the state-of-the-art of the field.

3.1 Introduction

Lying in the intersection of *Logic Programming* (LP) and *Machine Learning* (ML), Inductive Logic Programming (ILP) investigates methods for the generation of logic programs from examples and therefore inherits much of the theoretical framework of logic programming. In the following Sections we present all the logic programming concepts which are relevant to our work. We also describe the most important ILP methods and concepts, stressing what is more relevant to us.

3.2 Logic programs

A logic program, in the context of this work, is a set of clauses, i.e., *First Order Logic* formulas written in clausal form. In this Section we follow mainly the notation and terminology used by Hogger [46] and Lloyd [64]. In the latter one can find a detailed account of the theory of logic programming.

3.2.1 Syntax

A *clause* is a first order logic formula in clausal form:

$$\forall X_1, \ldots, X_s \, (L_1 \vee L_2 \vee \ldots \vee L_n)$$

where each L_i is a literal, $X_1, \ldots, X_s$ are all the variables occurring in the clause. A *literal* is an atom (positive literal) or a negated atom (negative literal). An *atom* is an expression of the form $p(t_1,t_2,\ldots,t_k)$, with $k \geq 0$, where p is the name of a predicate of *arity* k. Such a predicate name can also be represented by p/k. The t_i are the arguments of the atom. Each argument t_i is a *term*. A negated atom is of the form $\neg p(t_1,t_2,\ldots,t_k)$.

A term can be a *variable*, a *constant*, or a composed term of the form $f(t_1,t_2,\ldots,t_n)$, in which f is a *functor* with arity $n>0$ and the t_i are terms.

A clause can also be regarded as a set of literals $\{L_1, L_2, \ldots, L_n\}$. Another usual way of writing a clause is as an implication

$$A_1 \vee A_2 \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$$

where each A_i is a positive literal and each B_i is the atom of a negative literal. The sub-formula $A_1 \vee A_2 \vee \ldots \vee A_m$ is called the *head* of the clause or *consequent*. $B_1 \wedge B_2 \wedge \ldots \wedge B_n$ is called the *body* of the clause or *antecedent*.

Clauses can be classified according to their number of positive and negative literals. A clause with exactly one positive literal (exactly one literal in the head) and zero or more

negative literals is a *definite clause*. Any clause with more than one positive literal is an *indefinite clause*. A clause with no literals is called the *empty clause* and is denoted by a white square □. The empty clause represents contradiction: *false*←*true*.

A *recursive clause* is one in which at least one of its body literals has the same predicate as the literal in the head. A clause without variables is called a *ground clause*. Similarly, we have *ground literal* and *ground term*. A ground clause with a single positive literal is a *fact*.

A *logic program P* is a (possibly empty) set of clauses. The empty program is denoted by the symbol ∅. In order to represent logic programs we will use, for convenience, an identical notation as the one used in the logic programming language Prolog [116]. The symbols for disjunction (∨) and conjunction (∧) are replaced by commas, and clause ends with a period. However, we differ from Prolog notation in one aspect: the implication arrow (←) is used instead of Prolog's colon dash :-.

$$A_1, A_2, \ldots, A_m \leftarrow B_1, B_2, \ldots, B_n .$$

Variables are denoted by strings starting with an upper case letter (such as *X*, *Y*, *A*, etc.), and constants are denoted by strings starting with a lower case letter (such as *a*, *c*, *x*, etc.).

A *normal logic program* contains clauses with at least one positive literal. Each clause has the form

$$A \leftarrow L_1, \ldots, L_n.$$

where *A* is an atom and the L_i are literals. Each clause in a normal program defines the predicate *p/k* of atom *A*. The definition of a predicate *p/k* in a program *P* is the set of clauses in *P* which define *p/k*. A logic program containing only definite clauses is called a *definite logic program*.

Example 3.1: The logic program below has three clauses:

parent(X,Y)←father(X,Y).

parent(X,Y)←mother(X,Y).

ancestor(X,Z)←parent(X,Y), ancestor(Y,Z).

This is a definite logic program (therefore it is also a normal logic program) defining predicates *parent/2* and *ancestor/2*.♦

3.2.2 Semantics

A *Herbrand model* of a logic program *P* is, informally, a set of ground atoms which logically validate each clause of *P*. These ground atoms are elements of the *Herbrand base* of program *P*. The Herbrand base is the set of ground atoms which can be constructed using the predicates contained in *P* and any functors or constants belonging to the language.

A fact *q* is a *logical consequence* of a program *P* if all the models (Herbrand and non-Herbrand) of *P* are also models of *q*. That is denoted by

$P \models q$

A definite program (which excludes programs containing clauses with negative literals in the body) has a set of Herbrand models which are structured in a lattice according to the partial order relation $\subseteq$ between sets. The minimal element of this lattice is the *minimal (Herbrand) model*. The notation *MM*(*P*) denotes the minimal (Herbrand) model of the program *P*.

The minimal model of a definite program *P* corresponds to the set of ground atoms which are its logical consequences

$P \models q$ if and only if $q \in MM(P)$

In terms of denotation semantics, the *meaning of a (definite) logic program P* is *MM*(*P*). In other words, it is the set of ground logical consequences of *P*.

The semantics of a normal program *P* with negated literals in the body of at least one of its clauses is defined in terms of its *completion* denoted as *Comp*(*P*). The completion of a program is obtained by transforming its clauses into equivalencies, and adding special clauses defining an equality theory [64].

For a normal program *P*, instead of referring to the model of *P*, we refer to the model of *Comp*(*P*). However, to simplify the description of our work, we will say "the model of a program *P*" even if it is a normal program. We should however stress that many of the theoretical results obtained for definite programs are not valid for the generality of normal programs. We will identify those differences whenever it seems relevant.

The programs synthesized by our methodology are definite. The synthesis system may however have normal programs for background knowledge.

3.2.3 Derivation

A logic program is executed by posing *queries* to it. A query is a clause of the form $\leftarrow L_1,\ldots,L_n$ where each L_i is a literal. Basically, a query $\leftarrow q$ to a program *P* asks whether a fact *q* is a ground logical consequence of *P* or not, or if it is possible to assign certain values to the variables in *q* so that *q* is a ground logical consequence of *P* after replacing the variables by the corresponding values.

A query may *succeed* or *fail*. If it succeeds and the query contains variables, then a *substitution* (called *answer substitution*) is also part of the answer. A substitution is an application between a set of variables and a set of terms, and is represented as a set of variable/term pairs. Substitutions are usually denoted by Greek letters such as θ and σ. The process of substituting variables by terms is called *instantiation*.

Another fundamental concept is *unification*. Two atoms are unifiable when they can de made identical by substituting their variables by terms in a consistent way. A substitution which makes two atoms identical is called a *unifier*. Of all the unifiers of two atoms there

exists only one *most general unifier*. This corresponds, informally, to the substitution which minimally instantiates the two atoms.

Example 3.2: Atoms *f(a)* and *f(X)* are unifiable. The substitution $\theta = \{X/a\}$ is a unifier. By applying this substitution to the second atom we obtain the first one, $f(a) = f(X)\theta$. Substitution θ is also the most general unifier of the two atoms.♦

Resolution is an inference rule which enables the derivation of one clause *R* from two other clauses C_1 and C_2, called parent clauses. Clause *R* is the *resolvent*. Parent clauses must be complementary, i.e., for some literal A_1 in one of them there must be a literal $\neg A_2$ in the other so that A_1 and A_2 are unifiable. Therefore, if C_1 is the clause $A_1 \vee$ *More-Literals* and C_2 is $\neg A_2 \vee$ *Other-Literals*, resolvent *R* is (*More-Literals* $\vee$ *Other-Literals*)θ, where θ is the most general unifier of A_1 and A_2.

An answer to a query *Q* is derived from *P* using a *proof procedure*, an algorithm which applies a set of derivation rules to *P* and *Q* following a given strategy and constructing a *proof*. The proof procedure we use is *SLDNF-resolution*, which is used in the Prolog language interpreters. SLDNF-resolution is an extension of *SLD resolution*.

SLD-resolution works as follows. Given a query *Q* of the form

$$\leftarrow Q_1, Q_2, \ldots, Q_m$$

where each Q_i is a non-negated literal, and given a definite program *P*, SLD-resolution starts by selecting one of the literals from the query. Here, we will assume that the literal selection rule will always choose the leftmost literal, for it is the most common selection rule. In that case, the first literal to be chosen is Q_1.

Next, we choose a clause C_1 from the program *P*, so that the head of *C* can be unified with Q_1. Suppose that C_1 is of the form

$$A \leftarrow B_1, B_2, \ldots, B_n$$

Then, $A\theta_1 = Q_1\theta_1$, where θ_1 is the most general unifier of A and Q_1.

As such, we obtain the resolvent R_1

$$\leftarrow(B_1,B_2,\ldots,B_n,Q_2,\ldots,Q_m)\theta_1$$

After this first resolution step we will proceed in the same manner with resolvent R_1, selecting one clause C_2 from program P and obtaining a new unifier θ_2 and a new resolvent R_2. This process is repeated until we get the empty clause $\square$ as the resolvent.

An *SLD-derivation* of a program P from query $\leftarrow Q$ is represented by the sequence $D = ((R_1,C_1,\theta_1), (R_2,C_2,\theta_2), \ldots , R_n)$, where $R_1 = \leftarrow Q$, R_i is the resolvent of R_{i-1} and C_{i-1}, and θ_i is their most general unifier, for $1 \leq i \leq n$. A *refutation* of $\leftarrow Q$ from a program P is a derivation of P from $\leftarrow Q$ which ends with the empty clause ($R_n = \square$). By refuting $\leftarrow Q$ from P, we prove $Q\theta$ from P. Substitution θ is an answer substitution. We also say that $Q\theta$ is derivable from P.

The answer substitution θ to the initial query is obtained by composition of the most general unifiers θ_1, θ_2, etc. of the sequence in the derivation. When the empty resolvent is not derivable the query fails.

When a fact q is SLD-derivable from a program P we write

$$P \vdash q$$

The set of all the answer substitutions given by SLD-resolution to a query is obtained by searching exhaustively the space of SLD-derivations. The *SLD-tree* generated as a result this search has the starting query in its root, and each branch is one possible derivation of the program for the query. Each branch may either end with the empty clause (which corresponds to the answer), or on a non-empty clause which does not resolve with any clause of the program, or it can be an infinite branch. When all the branches of an SLD-tree for the query $\leftarrow Q$ are finite and none ends with clause $\square$, we say that the query finitely fails.

Example 3.3: Suppose we have the following program *P*:

descendant(X,Y)←son(X,Y).

descendant(X,Z)←son(X,Y),descendant(Y,Z).

son(alipio,antonio).

son(antonio,adriana).

The query ←*descendant(alipio,X)* is posed to program *P*. The answer θ={*X/antonio*} is given by the following derivation:

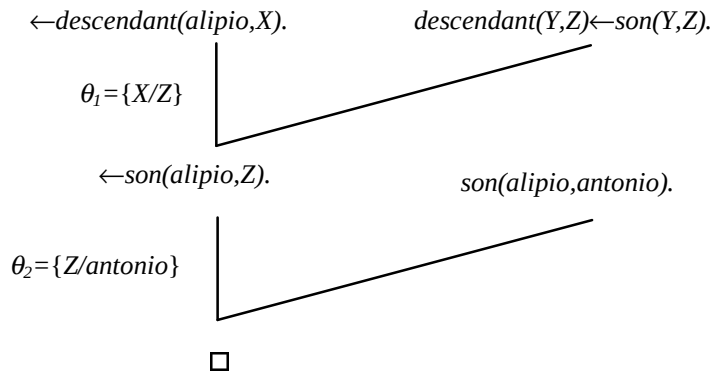

Figure 3.1: Derivation Graph

We have $P \vdash$ *descendant(alipio,antonio).*♦

Note that the set of facts derivable by SLD-resolution from a definite program *P* corresponds exactly to the minimal Herbrand model of *P*, i.e.,

$P \vdash q \Leftrightarrow P \models q \Leftrightarrow q \in MM(P)$

In other words, we can use SLD-resolution to determine which are the ground logical consequences of a program *P*.

To be able to deal with queries containing negated literals and normal programs, we need to extend the SLD resolution with the *negation as failure* rule. This is how we obtain

SLDNF-resolution. The negation as failure operator is usually denoted by *not* and is defined as follows: a query $\leftarrow not\ Q$ succeeds if and only if the question $\leftarrow Q$ finitely fails.

In terms of derivation, when a program P is being derived and a literal $not\ Q$ is found in the resolvent two situations may occur. The first possibility is that Q is derivable, and in that case $not\ Q$ cannot be resolved upon. The second possibility is that there is a finite SLDNF tree T for Q such that T has no successful branches (Q cannot de derived from P). We represent that derivation step as $(\leftarrow Lits_1 \wedge not\ Q \wedge Lits_2, T)$, $(\leftarrow Lits_1 \wedge Lits_2, C, \theta)$.

By imposing certain conditions to a program P, we can then relate the ground facts q derivable from P through SLDNF with the ground logical consequences of $Comp(P)$.

$$P \vdash_{\text{SLDNF}} q \Leftrightarrow Comp(P) \models q$$

Briefly, the conditions are as follows: no predicate p of P should be expressed directly or indirectly in terms of $not\ p$; all the variables of a clause should occur at least once in a non-negated literal; P should be *strict* in relation to any query q. For a definition of strict programs see [46]. Besides these conditions, the SLDNF-resolution should never select a negated literal that is not fully instantiated.

The relation defined between logic programs (sets of clauses of some language L) and facts (elements of the language L), through a set R of derivation rules, is called a *derivability relation*.

$$\vdash = \{<P,q> \mid P \subseteq L, q \in L, q \text{ is derivable from } P \text{ using } R\}$$

A proof procedure constructs one derivability relation. For readability, we will use the $\vdash$ symbol to denote both the SLD and SLDNF derivation.

Definition 3.1: Given a language L, a derivability relation $\vdash$, a program $P \subseteq L$ and a query $\leftarrow q$ such as $q \in L$, an *interpreter* for the language L is the operator,

$Int(P,\leftarrow q, \vdash) = \{ \theta \mid P \vdash q\theta \}$

♦

Each element of $Int(P,\leftarrow q, \vdash)$ is an answer substitution given by $\vdash$ for a query $\leftarrow q$ posed to P.

3.2.4 Types, input/output modes

A *type* corresponds to a non empty set of ground terms. This set is called a type domain or, simply a type. To every argument of a predicate we can associate a type. In the present work, this association is established through a *type declaration* of the form $type(p(type_1,\ldots,type_k))$. These declarations are given with the program specification for the predicate *p/k* (Section 4.3.1).

Argument types are used as a condition to be satisfied by the queries posed to the program and also by the answer substitutions [26]. An n-tuple of terms $(A_1,\ldots,A_n)$ is *compatible* with an n-tuple of types $(type_1,\ldots,type_n)$ if there exists a substitution θ such that $(A_1,\ldots,A_n)\theta \in (type_1\times\ldots\times type_n)$.

Example 3.4: We specify the type of arguments of *member/2* as $(X,Y)\in(integer\times list)$. This information is used as a pre-condition as well as a post-condition. As a pre-condition is used to filter the queries of *member/2*. Before a query $\leftarrow member(A,B)$ is executed, it is checked if (A,B) is compatible with $(integer,list)$. As post-condition it verifies if an answer substitution θ given is such that $(A,B)\theta \in (integer\times list)$. ♦

One advantage of type declarations is that they help the programmer to structure the logic programs he writes. Another one is that they allow the execution of these programs to be more efficient [26]. Type declarations are of interest to us mainly as a factor of efficiency in inductive logic programming (see Section 4.7 and [120]).

The *input/output modes* (or simply modes) of a predicate determine its possible uses [26,64]. For every predicate argument an input or output condition is defined. The input conditions should be verified before the execution of the logic program, whilst the output conditions should be verified after the answer substitution is obtained. The most simple input condition is "the argument should be a ground term". Another condition could be, for example, "the argument should be a variable". Output conditions are similar.

The input/output modes most frequently used in ILP determine which predicate arguments should be ground terms before execution [82,96,109]. For this reason, these arguments are called the *input arguments*. The remaining arguments are called *output arguments*. Here, an input/output mode declaration for predicate *p/k* is of the form

mode(p(M_1*, …,*M_k*)).*

where M_i is a plus sign '+' if the *i*-th argument is input, and a minus sign '-' otherwise.

Example 3.5: The mode of a predicate *p(X,Y)* can specify that this predicate should be invoked with the variable *X* instantiated. Variable *Y* may be instantiated or not. We call *X* an input argument and *Y* an output argument. The mode of predicate *p/2* is expressed as *mode(p(+,-)).*♦

For convenience we sometimes use the following notation. The '+' or '-' signs preceding the arguments of a literal in a clause, mean that these arguments have an input or output mode, respectively. This way, the literal *p(+a,+b,-c)* corresponds to the literal *p(a,b,c)*, with the input/output mode *p(+,+,-)*.

3.2.5 Integrity constraints

Integrity constraints are first order logic formulas of the form $A_1 \wedge \ldots \wedge A_k \rightarrow B_1 \vee \ldots \vee B_n$, where A_i and B_j represent literals. In general, integrity constraints are not representable by definite clauses. They are used in logic programming applications such as deductive databases [71,121] and inductive logic programming. In both cases, these special clauses

serve to prevent a given logic program from being updated in an undesirable way. In Chapter 7 we consider integrity constraints in more detail, particularly with respect to ILP applications.

3.3 The ILP problem

While in Logic Programming we proceed from programs to their logical consequences, in *Inductive Logic Programming* we start from the logical consequences and attempt to obtain the programs. The description of the logical consequences of the intended program is in the form of *positive* and *negative examples*. These examples are usually ground atoms[4]. Being represented by ground atoms, positive examples are like samples of the minimal model of the intended program. The limits of the model of the intended program are indicated by the negative examples: ground atoms which should not be logical consequences of the program.

The inductive task consists of finding a program *P* which is a hypothesis compatible with the given examples. This hypothesis is found within a *hypothesis language L* (also called *concept language*) which is a set of logic programs. We say that program *P* is *induced*, *synthesized*, or *learned*. The task of constructing a program inductively is called *induction*, *inductive synthesis*, *program synthesis* from examples or simply *machine learning* from examples. This multiplicity of terms is due to the fact that this problem is of interest to different communities within computer science and artificial intelligence. We will mainly use the designation *inductive program synthesis from incomplete specifications*. For commodity, we will sometimes say that *P* is 'the target/intended program' although in general there is a set of acceptable solutions for a synthesis problem.

[4] However, there are approaches which use non-ground clauses to represent positive and negative examples, as in [20,37,102], and our own work presented here.

As it happens with many other Machine Learning tasks, it is of utmost importance that the synthesis task of a program does not start from scratch. Having other predicates that can be used as auxiliary by the program is important. These are normally referred to as *background knowledge*.

The predicates in background knowledge can be defined either extensionally or intensionally. Background knowledge is *extensional* when it consists of a set of ground facts involving the auxiliary predicates. If auxiliary predicates are defined through program clauses which are not necessarily ground, then background knowledge is *intensional*.

The objective of ILP is generally presented as follows (De Raedt, Lavrac [24]*):*

Given

- a set of examples E (consisting of positive examples E^+, and negative examples E^-),
- background knowledge B,
- language L of logic programs,
- and a notion of explanation (a semantics),

find

- a program $P \subseteq L$ that explains the examples E relatively to B.

There are different notions of *explanation*. The most common is called the normal semantics of ILP. Another important notion of explanation is given through non monotonic semantics [22, 24, 36, 44]. In this work we will adopt normal semantics.

3.3.1 Normal semantics of ILP

A program P explains a set of examples $E = E^+ \cup E^-$ relatively to program B if

$P \cup B \models E^+$ (completeness)

and

$P \cup B \not\models e^-$ for all $e^- \in E^-$ (soundness)

Example 3.6:

Positive examples: {*descendant(alipio,antonio)*}
Negative examples: {*descendant(antonio,alipio)*}
Background knowledge: {*son(alipio,antonio), son(antonio,adriana)*}
Hypothesis: {*descendant(X,Y)←son(X,Y)*}

The conditions of completeness and soundness can be checked using the SLD-resolution (or the SLDNF-resolution if the clauses are not definite). Completeness is checked by verifying that all positive examples are entailed by hypothesis P together with background knowledge B (in this case there is only one positive example):

$P \cup B \vdash$ *descendant(alipio,antonio)*

The soundness condition is verified if no negative example is entailed by $P \cup B$.

$P \cup B \not\vdash$ *descendant(antonio,alipio)*

♦

Definition 3.2: A program P covers (intensionally) a fact e if $P \models e$. A program P covers (intensionally) a fact e relatively to a program B if $P \cup B \models e$.♦

Some ILP approaches use *extensional coverage*, a somewhat different notion that is computationally less demanding, but yielding different results.

Definition 3.3: A program P covers (extensionally) a fact e relatively to a model M if there exists a clause $C \in P$ ($C = H \leftarrow B$) and a substitution θ such that $C\theta$ is ground, $H\theta = e$ and $B\theta \subseteq M$. ♦

Example 3.7: Given program P

descendant(X,Z)←*son(X,Y),descendant(Y,Z).*
descendant(X,Y)←*son(X,Y).*

and the background knowledge

B = {*son(alipio,antonio),son(antonio,adriana)*}

P intensionally covers the example

descendant(alipio,adriana)

relatively to B. However, P does not extensionally cover the example.♦

In this dissertation the notion of intensional coverage will always be used, unless otherwise specified.

The conditions of completeness and soundness above presented, take into account only positive and negative examples. This scenario can be extended to include integrity constraints as a more expressive source of information, and particularly of negative information. An integrity constraint *Body*→*Head* is satisfied by $P \cup B$ if it is true in the minimal model of the program $P \cup B$. This can be checked transforming the integrity constraint into a query.

$P \cup B \nvdash$ *Body, not Head*

A set of integrity constraints is satisfied if each constraint in that set is satisfied. In Chapter 7, we will formalize these notions and present an efficient method for constraint checking.

Example 3.8: Let *I* be the integrity constraint

descendant(X,Y)∧*descendant(Y,X)*→*false*.

This constraint says that nobody is a descendant of one of his own descendants. Let *P* be the program

descendant(X,Z)←*son(X,Y).*
descendant(X,Y)←*son(Y,X).*

and *B* the background knowledge

B = {*son(antonio,adriana)*}

To check if program *P* is satisfied by constraint *I* we pose to the program the query

← *descendant(X,Y),descendant(Y,X).*

The query succeeds with *X*= *antonio* and *Y*= *adriana*. Therefore *P* does not satisfy *I*.♦

As will be seen in Chapter 7, both positive and negative examples can be expressed as integrity constraints. The conditions of completeness and soundness in the definition of the ILP problem can be replaced by the constraint satisfaction condition.

In our work the three conditions (completeness, soundness and constraint satisfaction) will be separatelly checked during induction. Soundness is checked for each tentative clause. Completeness is enforced by the synthesis strategy. Constraint satisfaction is checked with some degree of uncertainty.

3.3.2 Directions in ILP

The aim of ILP, as presented above, serves only as a starting point for a system which generates logic programs from examples. We will next refer to other aspects that can be considered when developing an ILP system:

- *Interaction*. An ILP system can be interactive or non-interactive. An *interactive* system asks questions to an *oracle* (usually the user) during the induction process. Systems MIS [109], CLINT [20] and SYNAPSE [37] are interactive. System SKILit presented here is non-interactive.

- *Noise*. The data supplied to the system can contain various types of incorrect information (e.g. an example provided as positive may in fact be negative). In this case, we say that the data is *noisy*. A system capable of handling noise must relax the conditions of completeness and soundness [61,62]. Our approach does not handle noise.

- *Predicate invention*. The auxiliary predicates defined within background knowledge may not be sufficient to find a satisfactory hypothesis. Some ILP systems avoid this limitation by inventing new predicates [57,114]. Here we do not consider predicate invention.

- *Single-predicate or multi-predicate learning/synthesis*. When a system accepts examples of different predicates, inducing definitions of various predicates simultaneously, it is said to perform *multi-predicate learning/synthesis* [25,100,109]. Otherwise, it is said to perform *single predicate learning/synthesis*. Here we concentrate mainly on single predicate learning. However, we show that our methodology also applies to multi-predicate synthesis.

- *Incrementality*. An incremental system has the ability of modifying an initial theory as new examples are presented. In the same situation, the non-incremental system discards the initial theory and restarts the induction of a new theory from scratch. This task is called *theory revision* [100,124]. Although our system is capable of eliminating clauses from the existing theory and adding new ones, here we concentrate mainly on the induction task.

3.4 Methods and concepts

Now that the ILP task is specified, we will see different approaches to construct a program *P* from examples and other sources of information.

3.4.1 The search in a space of hypotheses

As almost every other problem in artificial intelligence, finding an intended program can be reduced to a search problem. In this case, the *search space* is a set of programs in hypotheses language *L*. This space is structured by the relation of generalization between hypotheses.

Definition 3.4: (Muggleton, De Raedt [83]) : A hypothesis *A* is *more general* than a hypothesis *B*, if and only if, $A \models B$. Hypothesis *B* is said to be *more specific* than *A*. ♦

Starting from a set of initial hypotheses, the search is conducted by continuously applying generalization and/or specialization operators on the existing hypotheses until a stopping criterion is satisfied. A *generalization operator* produces a set of hypotheses $G_1, G_2, \ldots, G_n$ from a hypothesis *A*, and every G_i is more general than *A*. A *specialization operator* produces a set of hypotheses which are more specific than the initial one.

Structuring the search space according to a generalization relation enables filtering out many hypotheses. For instance, given a hypothesis *H*, a positive example *e*, and background knowledge *B*, if $H \cup B \not\models e$ then for no specialization *S* of *H* we have $S \cup B \models e$. This fact saves the effort of considering hypotheses which are more specific than *H* when trying to cover example *e*. Analogously, when a hypothesis *H* violates the soundness condition $H \cup B \not\models e^-$ for a negative example e^-, all hypotheses which are more general than *H* are also not sound. They can therefore be discarded.

The generalization relation based on the logical implication, corresponds to the most natural notion of generalization. However, logical implication poses some conceptual problems such as:

- Given two clauses C_1 and C_2 it is not a decidable problem to determine whether $C_1 \models C_2$.

- Two clauses C_1 and C_2 don't necessarily have a unique least general generalization under implication [47].

For that reason other generalization models have been proposed. Plotkin suggested *θ-subsumption* [92]. Buntine proposed *generalized subsumption* [16] that extends Plotkin's work. More recently, Idestam-Almquist brought forth *T-implication* [47], in an attempt to overcome some problems inherent in previous generalization models. Nevertheless, the model of *θ*-subsumption is the most frequently adopted in ILP algorithms. It is also the generalization model we adopt here and to which we give more emphasis.

3.4.2 The relation of *θ*-subsumption between clauses

The generalization relation between clauses is an important special case of the generalization between programs. Many ILP methods decompose the problem of searching in the general space of hypotheses into simpler search problems in the clause space.

Definition 3.5: (Plotkin [92]) A clause *C1 θ-subsumes* another clause *C2* if and only if there exists a substitution θ such that $C1\theta \subseteq C2$. ♦

The *θ*-subsumption relation is strictly weaker than the relation of logical implication [16]. If a clause *A θ*-subsumes a clause *B* then $A \models B$. The opposite is not true.

Example 3.9: Consider the next two clauses.

C1: p(X)←p(f(X)).
C2: p(X)←p(f(f(X)))

We have $C1 \models C2$ without having *C1 θ*-subsumes *C2*.♦

Definition 3.6: A clause *C1* is *θ-equivalent* to a clause *C2*, if and only if, *C1* θ-subsumes *C2* and *C2* θ-subsumes *C1*.♦

Definition 3.7: A clause *C* is *reduced* if it is not θ-equivalent to any subset of itself.♦

θ-subsumption between clauses induces a lattice in the set of reduced clauses. Any two clauses have a unique *least general generalization* (*least upper bound*), and a unique *most general specialization* (*greatest lower bound*) under θ-subsumption.

Within a given set of clauses, we may refer to *most specific clauses* and *most general clauses*. The clause *member(X,Y)* is most general among the clauses which define the predicate *member/2*. The clause □ (or *false←true*) is most general within any set of clauses. This clause corresponds to the empty set and therefore θ-subsumes any (other) clause. A most specific clause that covers an example *e*, relatively to a program *P*, is $e \leftarrow b_1, b_2, \ldots$ where the b_i are ground consequences of *P*. In this case, restrictions should be made so that the set of literals $\{b_1, b_2, \ldots\}$ becomes finite. The most specific clause is usually denoted by ⊥.

Given this generalization model, we now need operators which allow us to navigate in the set of clauses of the hypothesis language. We will see the refinement operator, a specialization operator relevant to our work, and a least general generalization operator. The latter will be described in less detail. Specialization operators allow us to move in the lattice of clauses from the most general to the most specific (*top-down approach*). Generalization operators make us go in the opposite direction (*bottom-up approach*).

3.4.3 The refinement operator (top-down approach)

Shapiro introduced the notion of a clause refinement operator under the θ-subsumption generalization model. Here, we give a more general definition following De Raedt and Lavrac [24].

Definition 3.8: An *operator* ρ associates to a clause C a set of clauses $\rho(C)$ called *refinements* of C. This is a set of specializations of C under θ-subsumption.♦

A typical refinement operator applies two sorts of transformations to specialize a clause:

1. variable instantiation;
2. joining a literal to a clause.

Example 3.10: The clause *member(A,[B|X])* can be specialized, for example, by instantiating *B* to *A*. We then obtain the refinement *member(A, [A|X])*. Another refinement can be obtained by adding a literal to the initial clause as in *member(A,[B|X])←member(A,X)*.♦

We can search for a the required clause by applying repeatedly a refinement operator. We start from the most general clause and then apply the refinement operator repeatedly to refinements. The search process terminates when one or more clauses are found to satisfy a given stopping criterion. This approach to the construction of a hypothesis is referred to as the top-down approach, since it goes from the most general clause to the more specific ones.

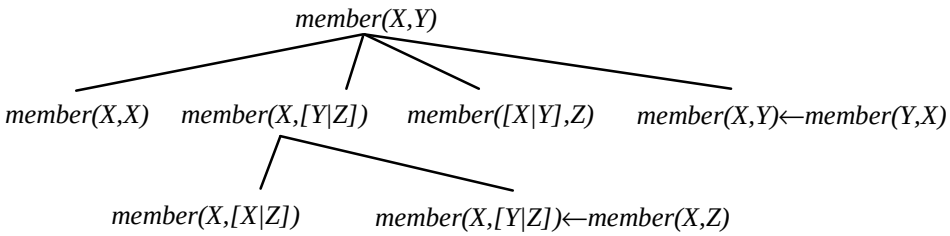

Figure 3.2: Part of one refinement graph [109].

A top-down search for a clause using refinement operators corresponds to a search in a refinement graph. A refinement graph is a directed acyclic graph, whose nodes are

clauses, and the root is the top clause. The branches of the graph correspond to specialization operations.

Various factors affect the size and shape of the search tree:

- *The top clause*. If we search for clauses to define the predicate *p/n*, then the most general clause is going to be $p(X_1,\ldots,X_n)$, where each X_i is a variable [96,109]. If we do not want to determine what the clause head predicate is, then we can start with clause *true*←*false* [22].

- *The refinement operator*. There are three main properties of a refinement operator, according to Muggleton and De Raedt [83]. The operator is *globally complete* if we can obtain any clause of the language by repeatedly applying the operator to the initial clause. The operator is *locally complete* if, for any clause *C*, $\rho(C)$ corresponds to the set of all of the most general specifications of *C*. Finally, the operator is *optimal* if it does not generate any clause more than once.

- *The stopping criterion*. The stopping criterion determines when to stop the search in the refinement graph. Normally, this criterion is defined in terms of the positive and negative examples. Shapiro's MIS system [109] stops the construction of a clause when it is specific enough not to cover negative examples. The stopping criterion can also demand that all the clause variables are linked [43]. Some systems use heuristics to define the stopping criterion [96].

- *The search method*. The order in which the nodes of the refinement graph are generated may also follow different strategies. The most frequently used search methods (see Section 3.4.5) are breadth-first [109], heuristic search (particularly greedy search methods [96,125]), and iterative deepening [22].

3.4.4 The lgg operator (bottom-up approach)

Under the θ-subsumption relation we can define the notion of the least general generalization of two clauses.

Definition 3.9: Clause *G* is a generalization of two clauses *A* and *B*, if and only if, *G* θ-subsumes *A* and *G* θ-subsumes *B*. ♦

Definition 3.10: A clause *G* is the *least general generalization* of clauses *A* and *B*, if and only if, for every generalization *G*' of *A* and *B*, *G*' θ-subsumes *G*. We write *lgg*(*A*,*B*)=*G*.♦

Example 3.11: The result of *lgg*(*p(a)*←*q(a), p(b)*←*q(b)*) is the clause *p(X)*←*q(X)* ♦

Plotkin, in his work about generalization under the θ-subsumption model [92,93], shows that the *lgg* of two clauses exists and it is unique (up to equivalence), and describes an algorithm to construct it.

More recently, Muggleton and Feng have popularized the lgg operator, by employing it in their GOLEM system [82]. In this system, the positive examples are first transformed into starting clauses which are most specific for the given predicate. Each of these starting clauses has a given positive example in the head. The body is a finite set of logical consequences of the background knowledge. By applying Plotkin's lgg operator, more general clauses are obtained from the starting ones. The most important contribution made by this work of Muggleton and Feng, was making Plotkin's original ideas efficient. This was mainly achieved due to the restrictions made to the hypothesis language. Other systems have, meanwhile, used the lgg operator [1,125].

3.4.5 Search methods

The search methods employed in ILP are basically the ones known from artificial intelligence. The *breadth-first search* method [59] is a type of brute force search where

the clause space is explored exhaustively. This is a complete search method, i.e., if an admissible solution exists in the search space then it will be found.

To perform the search, the breadth-first method keeps a *queue* of clause refinements. Initially the queue contains the top clause only. At each step, the method withdraws the first clause in the queue and expands it into a set of clauses. All the clause refinements resulting from the expansion are placed at the end of the queue. The expansion of a clause is made by applying a refinement operator.

Despite being complete, the method has the disadvantage of being inefficient (in terms of memory space and computational time). Its use is justified when the search space is made sufficiently small considering the available computational resources, and when other methods are not successful.

The *heuristic search* method is alternative to the brute force methods, in particular to breadth-first search methods. The heuristic search method computes for every candidate clause a value measuring how close it is from the objective. That value is calculated through what is called a heuristic function. Comparatively to the breadth-first method, the search is no longer blind: the most promising hypotheses are considered first.

The *hill-climbing* method chooses among all the clause refinements the one with the best heuristic value. The remaining refinements are discarded. The method has no backtracking (it is a greedy search method).

Although efficient, hill-climbing has the disadvantage of not being complete, since the search can follow a direction without any solution (a dead-end). Quinlan’s FOIL system [96] uses this search method. Other more sophisticated heuristic methods exist which can overcome some of the problems of the hill-climbing method[59].

3.4.6 Language bias

Any basis for restricting the size of the search space or for preferring one solution over another, apart from consistency with the observations, is called *bias* [73,115]. All the learning algorithms, including the ILP ones, employ some sort of bias to perform the search for solutions in a relatively efficient manner. The specific restrictions that are imposed to the hypothesis language are called *language bias*.

The hypothesis language can be constrained in many different ways. Here are some examples of language bias:

- *Admissible vocabulary:* The induced clauses can only involve predicates belonging to a pre-defined set. This set of predicates is called the *vocabulary*. In some approaches, the set of predicates admissible at a given stage is determined by other existing predicates, as in Russel's *determinations* [104].

- *Depth of terms*: Here, the restriction consists in limiting the depth of the terms that occur in the clauses It intends to capture the structural complexity of terms. The depth of variables and constants is 0. The depth of a term $f(t_1,\ldots,t_n)$, is $1+max(depth(t_i))$ [21,82].

- *Linked clauses:* A clause is *linked* if all its variables are linked. A variable is linked if it occurs in the head of a clause or in a literal that contains a linked variable (Helft [43]). This restriction avoids some potentially useless literals in the clause.

- *Depth of a variable :* The depth of variables occurring in program clauses can also be restricted. Let $p(X_1,\ldots,X_n)\leftarrow L_1,L_2,\ldots,L_r,\ldots$ be a clause. A variable occurring in the clause head $(X_1,\ldots,X_n)$ has a depth of 0. A variable V whose leftmost occurrence is in literal L_r has depth $1+d$, where d is the maximum depth of the variables in L_r which occur in $p(X_1,\ldots,X_n)\leftarrow L_1,L_2,\ldots, L_{r-1}$ [62].

- *Recursion:* Constructed programs may be non recursive. This is a very strong restriction, obviously not very adequate to Prolog program synthesis.

- *Determination*: Let $A \leftarrow L_1, L_2, \ldots, L_r, \ldots$ be a clause. A variable occurring in literal L_r is *determinate* if it has a unique valid substitution determined by the values of the variables in L_r occurring in $A \leftarrow L_1, L_2, \ldots, L_{r-1}$. The literal L_r is *determinate* if all its variables not appearing in $A \leftarrow L_1, L_2, \ldots, L_{r-1}$ are determinate. A clause is *determinate* if all its literals are determinate [62,82]. By imposing a limit j to the maximum arity of literals, and a limit i to the maximum depth of the variables in a determinate clause, we obtain *ij-determinate* clauses.

- *Types and input/output modes:* Type and input/output mode declarations are also useful for limiting the search space in ILP problems. The clauses of the hypothesis language which do not conform to the type or mode declarations may be filtered out [80,82,109].

Care must be taken when defining the appropriate bias. If the bias is strong, that is if it constrains the hypothesis language a great deal, the language may be incapable of representing a large family of concepts. However, the inductive system may be more efficient. Inversely, if the bias is weak (not very restrictive), then the system covers a larger spectrum of problems but at the cost of efficiency.

3.4.7 Declaring the language bias

In the light of the above, it seems that the language bias should be controlled by the user as much as possible, rather than being static. This type of bias which is defined by the user is called declarative bias.

The possibility of defining the language bias symbolically also has the advantage of enabling the ILP system to automatically change the hypothesis language whenever necessary. Therefore, the system may begin searching for a hypothesis in a relatively

simple language. If the search is unsuccessful the system moves to more complex hypothesis languages. This scheme is called *language shift* or *shift of bias* [20].

The simplest form of declaring language bias is by setting numerical parameters. This way, we can limit the number of clauses in a hypothesis, the number of literals in a clause, the number of variables of a determined type, the predicate arity, the depth of terms, etc. Such language biases are very common in ILP systems [82].

Meanwhile, other more sophisticated forms of describing the hypothesis language have been proposed. Wirth and O'Rorke [123] proposed *dependency graphs*, that illustrate the dependency relationships between literals. *Rule models* by Kietz and Wrobel [56], as well as the *clause schemata* by Feng and Muggleton [34] are higher order rules that represent sets of hypotheses. An example of a higher order rule is $P(X,Z) \leftarrow Q(X,Y),P(Y,Z)$. The symbols *P* and *Q* are variables which represent predicates. Substituting these variables by different predicate names we obtain different clauses. A possible substitution would give us

$$descendant(X,Z) \leftarrow son(X,Y),descendant(Y,Z).$$

Definite clause grammars or DCG, are also useful for describing the language bias. A DCG is a Prolog program written in a special notation for the encoding of grammars [88]. William Cohen, in his Grendel system [18] used the DCG formalism to define the admissible bodies of clauses. Klingspor [58] combined the approach of DCG with higher order rules. Instead of directly describing the hypothesis language his grammars define a set of higher order rules which can be instantiated. The clauses of the hypothesis language are obtained by instantiation. Our own induction methodology described here uses the DCG formalism to represent the program knowledge useful for program synthesis. Another possibility for language bias description was presented by Bergadano [5].

Birgit Tausend joined in a single formalism many different forms of bias representation. Her language MILES-CTL [119] allows the description of sets of clauses by using structures called *clause templates*. Inside these structures we can use predicate variables, define types of predicates and arguments, restrict the arity of predicates, etc. Using the MILES-CTL Tausend compares the impact of different language biases on a set of test cases. [120].

We have identified another sort of declarative bias that is useful to the synthesis process [14]. When the user is able to describe how an algorithm works on a particular example, even if in an inaccurate and vague way, the system can exploit that information in order to reduce the search effort. In Section 4.5.1 we describe how to represent this information using what we call *algorithm sketches.*

3.5 State-of-the-art of ILP

3.5.1 Origins of ILP

Nowadays, ILP is a very active research field, and occupies a significant position within machine learning [84]. Earlier learning from examples used zero-order languages (conditions in the form of attribute-value pairs, decision trees) to represent the hypotheses, or very restrictive forms of predicate calculus [66].

The works of Banerji [3], Plotkin [92,93], Michalski [67], Vere [122], Brazdil [13] and Sammut [107], amongst others, proposed approaches to make hypothesis languages more expressive. The motivation was to make algorithms for learning from examples more widely applicable [106]. However, as the hypothesis language became more expressive, the learning algorithms had to search through larger hypotheses spaces and, in consequence, the design of these became a challenge. A unifying principle or theory was also missing.

One such theory was proposed by Shapiro [109], who used definite clauses to represent the hypotheses and a small set of operators for the generation of plausible hypotheses within his MIS system. Towards the end of the eighties and early nineties, logic programming was adopted as the basis of logical approaches to machine learning from examples. Muggleton coined the term *Inductive Logic Programming* [77]. Various other systems emerged.

3.5.2 Some ILP (and alike) systems

Shapiro's MIS system is geared towards algorithmic debugging of logic programs. Logic program synthesis from examples can be regarded as a special case of this more general problem. For each session with MIS, some positive and negative examples must be supplied initially. More examples get requested by the system during the inductive process. Besides the examples, the system accepts type and input/output mode declarations of the involved predicates. Dependency declarations between predicates are also given to the system. Background knowledge is defined intensionally.

The systems GOLEM, by Muggleton and Feng [82], and FOIL, by Quinlan [96, 97, 98], were quite successful due to their relative efficiency and some practical problems to which these systems were applied. The GOLEM system induction engine is based in the *lgg* operator of Plotkin [92], which was already described here (Section 3.4.4). The system performs an incomplete bottom-up search: it constructs maximally specific clauses from randomly chosen examples and then applies the *lgg* operator to obtain more general clauses. The clauses which cover more positive examples and less negative examples are chosen.

The FOIL system constructs each clause following a top-down approach. The top clause is the most general clause (e.g. *member(X,Y)*). The system uses the hill-climbing search method, and the heuristic function is defined in terms of an information-theoretical measure based on the number of covered positive and negative examples. Constructed

clauses are appended to a candidate program following an AQ-like covering strategy [67,70].

Systems GOLEM and FOIL accept ground positive and negative examples supplied by the user. In addition to that, input/output mode declarations and dependency declarations for every predicate are given. Both systems are non-interactive and non-incremental. In both cases background knowledge is extensionally defined.

System Progol, by Muggleton [80], searches for every clause using a bottom-up approach similar to GOLEM's. It starts with a most specific clause and constructs one of its possible generalizations. The head of the starting clause is a positive example. The body is a subset of the model of the background knowledge. The search for the generalization is guided by an A^* like method [59]. Progol is relatively efficient when compared to GOLEM and FOIL. It allows an intensional representation of background knowledge.

CLINT [20] is an interactive and incremental system that constructs a theory from ground positive and negative examples and background knowledge. Given a clausal language L, CLINT takes each uncovered positive example e and constructs a set S of initial clauses covering e which are maximally specific in L (according to the θ-subsumption relation). These clauses must not cover any negative example. Afterwards, each clause $C \in S$ is maximally generalized by removing literals from the body of the clause. Before removing a literal, the system queries the user about the truth value of an example which is covered by the tentative clause but not by C. If all new examples are positive, the generalization step is accepted, otherwise it is rejected. The negative examples obtained in the process of generalizing a clause are used to detect and remove possibly incorrect clauses. The user is again queried in the process.

The SYNAPSE system [38] of Pierre Flener belongs to a different class. It is exclusively devoted to automatic programming tasks. The system synthesizes programs from ground examples and from *properties* (correct but incomplete clauses), and is a hybrid of

different approaches to program synthesis. Calling it an ILP system is a little misleading. In SYNAPSE we can find deductive synthesis, knowledge based synthesis and learning from examples [37].

The synthesis is guided by a *scheme* that encodes a particular programming strategy (divide-and-conquer, generation-and-test, producer-consumer, etc.). The program is constructed by transforming this scheme. SYNAPSE interacts with the user, to avoid exponential search. The SYNAPSE system does not use auxiliary programs supplied by the user (background knowledge), but performs predicates invention.

System CRUSTACEAN [1] is a follow-up of system LOPSTER [60] and induces logic programs of the form

$p(Tb_1,\ldots,Tb_n)$.
$p(Th_1,\ldots,Th_n) \leftarrow p(Tr_1,\ldots Tr_n)$.

where each Tx_i is a term. The base clause and the recursive clauses are constructed by structural decomposition of the given ground positive examples. Ground negative examples are also given and are used to eliminate overgeneral candidate programs. Decomposing an example consists of finding all the possible *subterms* of its arguments. For instance, suppose we have the positive example *last_of(a,[c,a])*. The first argument can be decomposed into subterm *a* only. The second argument *[c,a]* can be decomposed into *[c,a]*, *c*, *[a]*, *a* and *[]*. Each subterm is obtained by applying a sequence of decomposition operators to the initial term. This sequence is named the *generating term*. The number of times the generating term is applied is called *depth*. Term *[a]*, for instance, is obtained from *[c,a]* by the generating term *pair(2)*, i.e., the function that returns the tail of the list. The depth is 1. When the subterm is obtained by no decomposition, the generating term is *none*. CRUSTACEAN obtains all the possible decompositions of the example by combining all possible decompositions of its arguments. One possible decomposition of the example *last_of(a,[c,a])* is *last_of(a,[a])*. It is obtained by the combination of generating terms (*none, pair(2)*) at depth 1.

Now suppose there is another positive example *last_of(b,[x,y,b])*. One of the decompositions of this example is *last_of(b,[b])*. The corresponding generating terms are *none* for the first argument, and *pair(2)* for the second. However, *pair(2)* must be applied twice (depth 2).

CRUSTACEAN can now combine the two decompositions of the examples, since they have the same generating terms (*none*,*pair(2)*). The result of the combination is a program. The base clause is the *lgg* of the atoms which result from the application of the generating terms to the examples. In other words, *lgg*(*last_of(a,[a])*, *last_of(b,[b])*), i.e.

last_of(A,[A]).

To obtain the head of the recursive clause, we apply the generating terms to the examples 0, 1,…, *n-1* times where *n* is the respective depth. The resulting atoms are *last_of(a,[c,a])* for the first example, and *last_of(b,[x,y,b])*, *last_of(b,[y,b])* for the second example. The head of the clause is the *lgg* of these three atoms. The recursive literal is obtained by applying the generating terms to the head.

last_of(A,[B,C|D])←last_of(A,[C|D]).

Obviously CRUSTACEAN does not find the right combination of generating terms of the examples directly. All the different generating terms used to obtain all the subterms of all the arguments of all the examples must be found. After that, the system constructs all the possible combinations of the generating terms of the arguments for each example. The combinations of different examples are then matched in all possible ways. Each match is either discarded because of incompatibility of generating terms or results in a program. Programs are then filtered. Redundant programs, infinitely recursive programs and programs covering negative examples are not considered. The remaining programs are the answer.

Because of the very restrictive language bias, CRUSTACEAN is not able to exploit any sort of background knowledge.

System	***Collection of examples***	***Strategy***	***Background knowledge***	***Example of applications***
MIS	interactive, incremental	complete search	intensional	prog. synthesis
GOLEM		heuristic selection of hypotheses with random generation of seeds. Uses lgg.	extensional	biology, mesh design, quantitative models
FOIL		covering AQ-like strategy. top-down construction of clauses. hill-climbing.	extensional	prog. synthesis
Progol		covering AQ-like strategy.	intensional	biochemistry
CLINT	interactive, incremental	bottom-up construction of clauses	intensional	prog. synthesis knowledge base updating autonomous agents
SYNAPSE	interactive	scheme transformation.	none invents predicates	prog. synthesis
CRUSTACEAN		term decomposition	none	prog. synthesis

Table 3.1: Main characteristics of some important ILP systems.

The systems referred above represent only a selection of the state-of-the-art in ILP. Other systems are also of interest. This is the case of systems such as CHILLIN [125], CLAUDIEN [22], FOCL [110], FORCE2 [12], FORTE [100], ITOU [102], MOBAL [76], SMART [74], TIM [49], WiM [95], etc. However, our intention here is not to make an exhaustive description of these systems.

3.5.3 Applications

Most ILP applications fall either in the area of knowledge extraction and discovery or program synthesis. As for the applications in the area of knowledge extraction and discovery, GOLEM, for instance, has been applied to the problems of qualitative model construction [11], construction of temporal models for satellite maintenance operations [33], protein structure prediction [77], and mesh design [29]. Progol has already been applied for knowledge extraction in biochemistry [85]. The results produced by Progol were published in a biochemistry scientific journal [86]. Other systems also had practical

applications, as is the case of MOBAL [75] , CLAUDIEN [22], FORTE [100] and FOCL [32].

3.5.4 Inductive program synthesis

If in the field of knowledge extraction and scientific discovery ILP is already a useful tool, the same cannot be said with respect to program synthesis from examples (inductive synthesis). In this field, there are still important problems to be solved before we have a true practical application. The aim of our work is to move forward in the direction of using inductive tools to aid in the development of small programs.

In this Section we informally show illustrative results of systems which are representative of what has been achieved in the field of inductive program synthesis. The systems referred to are MIS, GOLEM, FOIL, SYNAPSE and CRUSTACEAN.

Let us first see an example of an MIS session as given by Shapiro [109]. The task is to synthesize predicate *isort/2* which sorts a list using an insertion strategy. A definition for *isort/2* is synthesized as follows:

isort([X|Y],Z)←isort(Y,V),insert(X,V,Z).
isort([],[]).

The auxiliary predicate *insert/3* is also synthesized.

insert(X,[],[X]).
insert(X,[Y|Z],[X,Y|Z])←X≤Y.
insert(X,[Y|Z],[Y|V])← insert(X,Z,V),Y≤X.

The session is reported in eight (!) pages which are mainly filled with information given by the system describing the current situation (these descriptions must be checked by the user), as well as with the queries asked to the user and corresponding answers. A summary of the session indicates that 30 facts on *isort/2* and *insert/3* were necessary for the synthesis. As for CPU time, 36 seconds were needed.

In the field of program synthesis from examples, the GOLEM system was successful in the induction of predicates, such as of *member/2*, *reverse/2*, *multiply/2* and *qsort/2*, but only when the examples were carefully chosen.

The recursive clause of the predicate definition *qsort/2* (*quick sort*) is a classical test for a system performing synthesis from examples:

```
qsort([],[]).
qsort([A|B],[C|D])←
        partition(A,B,E,F),
        qsort(F,G),
        qsort(E,H),
        append(H,[A|G],[C|D]).
```

This clause has two recursive literals, which makes it problematic for some synthesis strategies. Furthermore, the clause has 4 literals in the body (6 if functors are not used), a relatively large number of variables (8), and some of them have a depth of 3. GOLEM generated the definition of '*quick sort*' from 15 well chosen examples, in about one hundredth of a second. The background knowledge contained 84 facts on *partition/4* and *append/3*. Obviously these results are not guaranteed if other examples are used.

The FOIL system was evaluated in [97] by its authors. The task for this test consisted in synthesizing a series of predicates taken from Bratko's "*Prolog for Artificial Intelligence*" [10]. As an example, we show the definition generated for *reverse/2*:

```
reverse(A,B)←A=B, dest(A,C,D), sublist(A,C).
reverse(A,B)←
        dest(A,C,D),reverse(D,E),
        append(F,D,A),append(E,F,B).
```

This definition was synthesized from 40 positive examples and 1561 negative examples (see Appendix A for definitions of auxiliary predicates such as *append/3*, etc.). The examples given are all the examples that involve lists of size 3 or less. Although FOIL needs a large number of examples to generate a program, it is robust in the presence of redundancy in the background knowledge.

System Progol synthesizes a definition of ‘quick sort’ in less than a second, given 11 positive and 12 well chosen negative examples. In [80] we can find a summary of the results obtained from Progol in inductive synthesis.

The SYNAPSE system can synthesize programs as hard as ‘*insertion sort*’, although yielding a different definition from the one obtained with MIS. Given 10 positive examples, the following 3 properties

isort([X],[X]).
isort([X,Y],[X,Y])←X≤Y.
isort([X,Y],[Y,X])←Y>X.

and specific programming knowledge relative to this problem, a definition of *isort/2* is generated. During the synthesis process a definition for the predicate *insert/3* is invented [37, pp.209]. Thus the user does not have to provide auxiliary predicates required to synthesize this predicate.

The system CRUSTACEAN can synthesize recursive programs with functors without auxiliary predicates. Every program has a base clause and a recursive clause. Here is an example:

split([],[],[]).
split([A,B|C],[A|D],[B|E])←
split(C,D,E).

CRUSTACEAN can generate this program from 2 positive examples and 4 negative ones, without any further information. However, the system is restricted to a very limited hypothesis language. The strategy used by the system is very robust with respect to the choice of examples. In other words, the two examples given do not have to be carefully chosen in order to synthesize the recursive program shown above.

3.5.5 Problems and limitations

- *Intensional background knowledge*. Systems GOLEM and FOIL only accept extensional background knowledge. Extensional representation of the predicate in the background knowledge provides greater efficiency. However, the construction and maintenance of large background knowledge is difficult [79]. Some systems (CRUSTACEAN, SYNAPSE) do not even allow the use of background knowledge.

- *Recursive program synthesis from sparse sets of examples*. Progol, as well as GOLEM and FOIL, have problems in synthesizing recursive logic programs from a set of relatively small positive examples. Quinlan points out that the synthesis of *member/2* is not robust. In one experiment, it was observed that when 25% of the positive examples were eliminated at random the induced program was still correct, but contained three redundant clauses [97].

- *Use of generic programming knowledge*. The present ILP systems, with few exceptions, perform a blind search for the target program. Few take advantage of existing knowledge about programming. One exception is the SYNAPSE system, which constructs the clauses following the strategy of divide-and-conquer. Even this system does not allow the definition of new strategies without changing the code of the system itself. In Section 4.5.1.1 we describe clause structure grammars: a formalism to represent generic programming knowledge which enables to overcome this shortcoming.

- *Use of specific programming knowledge*. If the user has some notion, however incomplete, about the strategy that a particular program to be synthesized should follow, he should have the opportunity of giving that information to the system. The algorithm sketches presented in Section 4.5.1 allow this sort of information to be conveyed to system SKILit.

- *Over-generalization*. The excessive number of negative examples many ILP systems need in order to induce the target programs is a problem that has already been

recognized by this research community. The strategies that have been proposed are, in our view, unsatisfactory. The user of a program synthesis system should be able to represent the intended negative information in a compact way. Integrity constraints allow this compact representation, but creates great efficiency problems. In Chapter 7 we propose an efficient algorithm that allows the use of integrity constraints in the context of ILP.

3.6 Summary

In inductive synthesis of logic programs, ILP is a promising research area, but more work is required before the technology is useful in practical applications. It is however taking large steps in that direction. Before that happens, some problems have to be solved, such as the synthesis of recursive programs from sparse sets of positive examples, the effective use and representation of generic programming knowledge and specific programming knowledge, as well as the use of integrity constraints for the representation of information usually given to the system through negative examples.

In Chapters 4, 5 and 7 we address these problems and propose a program synthesis methodology which attempts to overcome some of the current limitations of inductive approaches to program synthesis from examples.

4. An Approach to Inductive Synthesis

This chapter presents an approach to logic program synthesis from incomplete specifications. System SKIL is introduced. We describe the information that is given to the system, and define the class of synthesizable programs. The synthesis process and its main algorithms are described.

4.1 Introduction

In this Chapter we describe the methodology on which system SKIL is based. This is an inductive logic programming system geared towards the synthesis of logic programs (or simply programs) from examples of their behaviour. In terms of program synthesis we can see SKIL as a synthesis system from *incomplete specifications*.

The starting point is an incomplete description of a predicate *p/k*. The aim is to synthesize a program *P* defining *p/k*. This description is called a specification and consists of *positive* and *negative examples* of that predicate, *integrity constraints*, *input/output mode declarations* and *type declarations*. From this data the system constructs a program that generalizes the positive examples, and that is consistent with

the negative examples and integrity constraints (Chapter 7). The program produced by SKIL consists of definite clauses with no functors.

Another important element of the process is the *background knowledge* (BK). This is a logic program that defines auxiliary predicates that can be used in the definition of the predicate to be synthesized. Although it is not regarded as a language bias, the background knowledge also affects the set of synthesizable clauses. It has a determinant role in the selection of literals due to the clause construction strategy employed by SKIL.

Besides the specification and the background knowledge, the SKIL system exploits other sources of information that affect the synthesis process. It is the case of the *algorithm sketches* and the *clause structure grammar* (CSG). The clause structure grammar can be seen as a way of defining the language bias, for it defines the set of clauses synthesizable by the system.

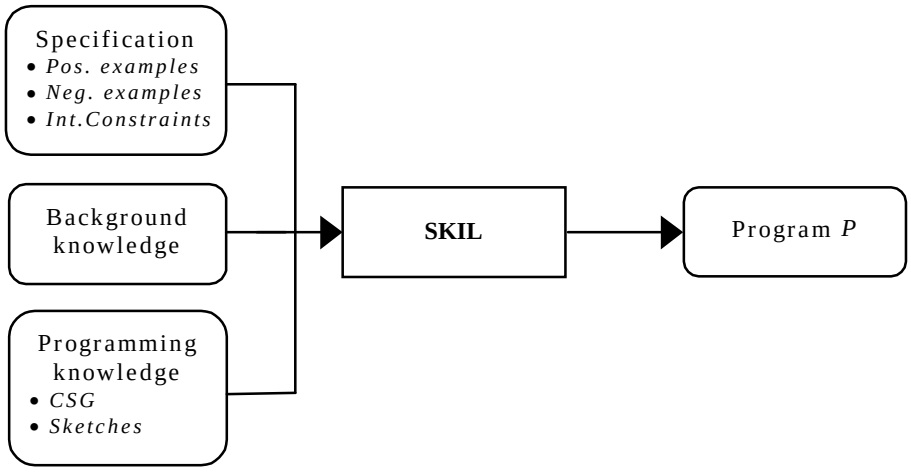

Figure 4.1: Framework of the SKIL system

4.2 Overview

We first describe the input of SKIL. What is a specification (Section 4.3), what can be background knowledge (Section 4.4) and how is programming knowledge represented

(Section 4.5). Within the programming knowledge Section we define algorithm sketches and explain the role of clause structure grammars. In Section 4.6 we characterize the programs SKIL can synthesize.

The process of synthesizing a logic program is described in detail in Section 4.7. There we describe the algorithms for program construction (SKIL) and clause construction. We present the sketches refinement operator and the notion of relevant sub-model. We also describe the depth-bounded interpreter used in the interpretation of background knowledge and constructed programs, and how the clause structure grammars are used within the refinement operator. The Section ends with a description of type checking in SKIL.

In Section 4.9 we show a synthesis session with SKIL and in the remaining three Sections we discuss limitations of the methodology, related work and give a brief summary of this Chapter.

4.3 Specification

The *specification* supplied to the SKIL system is *incomplete*. The program behaviour that is not described in the specification is inferred. The specification describes one single predicate *p/k* to be defined as program *P*.

Given predicate *p/k*, a specification is defined as a tuple (T,M,E^+,E^-,IC) where

- T is the type declaration for predicate *p/k*;
- M is the input/output mode declaration for *p/k*;
- E^+ is a set of positive examples of *p/k*;
- E^- is a set of negative examples of *p/k*;
- IC is a set of integrity constraints restricting *p/k*.

The Figure 4.2 below, shows the typical format of a specification given to SKIL. The notation has a Prolog-like syntax: mode and type declarations, examples, and integrity constraints are represented as clauses.

mode(p(m_1, …,m_k)).
type(p(t_1,…,t_k)).

% positive examples
p(…).
…
p(…).

% negative examples
–p(…).
…
–p(…).

% integrity constraints
p(…),…,q(…)→r(…),…,s(…).
…

Figure 4.2: Typical format of a specification for predicate *p/k*.

4.3.1 Objective of the synthesis methodology

Given background knowledge *BK* and a specification (T,M,E^+,E^-,IC) describing predicate *p/k*, SKIL constructs program *P* defining *p/k*. The program has, ideally, the following properties:

- All the positive examples are covered:

  $P \cup BK \vdash E^+$

- No negative example is covered:

  $P \cup BK \nvdash e^-$ for all $e^- \in E^-$

- The constructed program satisfies the integrity constraints (this condition is checked with some degree of uncertainty due to the Monte Carlo strategy employed, as we will later see in Chapter 7).

  $P \cup BK \not\vdash (Body, not\ Head)$ for all $I \in IC$, I has the form $Body \rightarrow Head$.

4.3.2 Examples, modes, types, integrity constraints

The *positive examples* given to SKIL are ground atoms. The *negative examples* are ground atoms marked with a '–' sign. The *mode declaration* of a predicate *p/k* assigns to each one of the *k* arguments an input or output direction. The input arguments are marked with a '+' sign, and the output ones a '–' sign.

A positive example of the predicate *reverse/2*, that reverses a list, can be *reverse([2,1],[1,2])*. This positive example determines that the program to synthesize should output that the reverse of list *[2,1]* is list *[1,2]*. A negative example of the same relation is *–reverse([0,3],[0,3])*.

The input/output mode declaration is

mode(reverse(+,–)).

The meaning of this input/output declaration is that a query to the program which defines the predicate *reverse/2* must have the first argument instantiated before being executed, as in *←reverse([2,4,3],X)*.

The *type declaration* associates to each argument an identifier that represents the assigned type. The types considered here include lists (*list* identifier), integers (*int* identifier), etc. (Appendix B). In the case of predicate *reverse/2*, the type declaration is *type(reverse(list,list))*. The type declarations facilitate the process of induction, but they are optional. In Figure 4.3 we see an example of a specification.

```
mode( reverse(+,–) ).
type( reverse(list,list) ).

% positive examples
reverse([],[]).
reverse([1],[1]).
reverse([1,2],[2,1]).

% negative examples
–reverse([],[1]).
–reverse([1,2],[1,2]).
–reverse([1,2,3],[2,1,3]).

% integrity constraints
reverse([A,B],[C,D])-->A=D.
reverse([A,B],[C,D])-->B=C.
```

Figure 4.3: Example of a specification for the predicate *reverse/2*.

Integrity constraints are non-ground clauses containing negative information just like negative examples do. Every negative example can be transformed into an integrity constraint. To make the description of the method clearer, we will separate the description of how negative examples and integrity constraints are handled. The latter issue will be described in Chapter 7.

4.4 Background knowledge

The background knowledge supplied to the SKIL system is a Prolog program that defines the auxiliary predicates which can be invoked by the program to synthesize. Background knowledge clauses can contain functors and negation. Figure 4.4 shows the sort of auxiliary programs that can be found in the background knowledge.

```
addlast([],X,[X]).
```

addlast([A|B],X,[A|C])←
addlast(B,X,C).

null([]).

dest([A|B],A,B).

const([A|B],A,B).

Figure 4.4: An example of background knowledge

Among the predicates defined in the background knowledge, the user can indicate which are the *admissible predicates* for a given synthesis task. This is done through a declaration that is given to the system, jointly with the specification. Let us see an example.

adm_predicates(reverse/2, [const/3,dest/3,null/1,addlast/3,reverse/2]).

The above declaration indicates that the system can induce a definition for the predicate *reverse/2* with clauses involving predicates *const/3*, *dest/3*, *null/1*, *addlast/3* and *reverse/2*, and only these predicates. The admissible predicate declaration defines the *vocabulary* for the synthesis task.

4.5 Programming knowledge

Besides the information contained in the specification and the background knowledge, SKIL employs other sources of auxiliary knowledge, such as *sketches*, which contain specific knowledge for every synthesis task and the *clause structure grammar*, which contains generic programming knowledge. This body of information is what we call *programming knowledge*.

These elements are obviously not considered part of the specification itself. They should instead be regarded as tools used to accomplish the synthesis task. While the examples

and integrity constraints indicate *what* is intended to be synthesized, the sketches and grammars indicate *how* the synthesis should or can be done. This distinction between the 'what' and the 'how' of the synthesis process has been pointed out in Chapter 0.

4.5.1 Algorithm sketches

The user of a synthesis system may know which particular predicates are involved and how those predicates contribute to the derivation of a given positive example. If this sort of knowledge exists, then it is of interest that the synthesis system is able to exploit it. This knowledge is communicated to the system through an *algorithm sketch*. The SKIL system is able to exploit algorithm sketches supplied by the user [14]. We should stress, however, that algorithm sketches are not mandatory input.

4.5.1.1 What is an algorithm sketch?

Informally, an algorithm sketch represents the explanation of a positive example in terms of relational links from the input to the output arguments of the example. Formally, an *algorithm sketch* relative to a program P is a ground clause whose head is a positive example of a predicate *p/k* defined in P, and the body contains literals which explain the output arguments of the example from the input arguments. When part of the explanation is not known, the arguments are linked by special literals called *sketch literals*. The remaining literals, involving admissible predicates are called *operational literals*. The predicates used in sketch literals (*sketch predicates*) start with the $ character. These sketch predicates also have an input/output mode.

Definition 4.1: Let α be a set of literals, t is a *directionally linked term* in α with respect to a set of terms T if and only if $t \in T$ or t is an output argument of some literal $L \in \alpha$ and all the input arguments of L are directionally linked in α with respect to T. ♦

Please note that in the following we use a clause-like notation for representing sets of literals. Therefore, the sequence $L_1, L_{2,\ldots}, L_n$ represents the set of literals $\{L_1, L_{2,\ldots}, L_n\}$.

Example 4.1: The term *e* is directionally linked with respect to {*a*,*b*} in the set of literals *p(+a,–c),q(+b,–d),r(+c,+d,–e)*. The link is graphically represented in Figure 4.5.

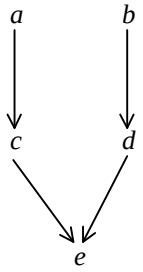

Figure 4.5: Linking terms {*a*,*b*} to term *e*.

In the same set of literals we can find other links. For example, the term *b* is directionally linked with respect to {*b*}.♦

Definition 4.2: A set of literals α is a *relational link* from a set of terms T_1 to a set of terms T_2 if and only if every term *t* occurring in α is directionally linked in α with respect to T_1 and every term in T_2 occurs in α.♦

Example 4.2: A relational link links terms T_1 to terms T_2 and contains no literals with terms that are not linked with respect to T_1. The set α = *p(+a,–b),q(+c,–d)* is not a relational link from {*a*} to {*d*} because *c* is not directionally linked in α. However it is a relational link from {*a*,*c*} to any subset of {*a*,*b*,*c*,*d*}.

The set of literals *p(+a,–c),q(+b,–d),r(+c,+d,–e)* is a relational link from {*a*,*b*} to any subset of {*a*,*b*,*c*,*d*,*e*}.♦

Definition 4.3: A term *t* is *directionally linked* in a clause $H \leftarrow \beta$, where β is a set of literals, if and only if there is a relational link $\alpha \subseteq \beta$ from the input arguments of *H* to *t*.♦

Definition 4.4: A clause $H \leftarrow \alpha$ is a *directionally linked clause* if all output arguments of *H* are directionally linked terms in α with respect to the set of input arguments of *H*.♦

Definition 4.5: An *algorithm sketch* is a directionally linked ground clause of the form $H \leftarrow L_1,L_2,..,L_n$ with $n \geq 1$, where H is a positive example of some predicate to be defined, and the literals $L_1,L_2,\ldots,L_n$ can be either operational literals or *sketch literals*. ♦

Sketch literals are employed to link arguments that otherwise would remain unlinked. Syntactically they are distinguished by predicate symbols like *$Px*, where *x* is a positive integer.

Example 4.3:Let *rv([3,2,1],[1,2,3])* be a positive example of predicate *rv(+,–)*. The following clause is a sketch.

r v(+[3,2,1],–[1,2,3])←
$P1(+[3,2,1],–3,–[2,1]),rv(+[2,1],–[1,2]),$P2(+3,+[1,2],-[1,2,3]).

This sketch involves two sketch predicates *$P1* and *$P2*, and one operational predicate *rv/2*. It can be seen as an explanation of how to reverse list *[3,2,1]*: "first obtain *3* and *[2,1]* (it is not described how), reverse *[2,1]*, and combine the result of the latter with *3* to obtain *[1,2,3]* (again, somehow)".

In the above sketch, the input list *[3,2,1]* is linked to *[1,2,3]*. Figure 4.6 shows a graphical representation of the sketch.

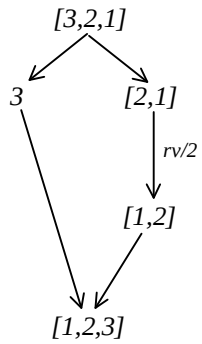

Figure 4.6: Graphical representation of one sketch.

♦

4.5.1.2 Positive examples are black box sketches

Any positive example can be regarded as a sketch containing no information about how the output arguments can be obtained from the input ones. The link between input and output arguments is then done by a single sketch literal whose only purpose is to make the missing connections explicit.

Definition 4.6: When the body of the sketch contains just one sketch literal, the sketch is called a *black box sketch*. The black box sketch associated to a positive example $p(t_1,\ldots,t_k)$ has the form

$$p(t_1,\ldots,t_k) \leftarrow \$P(t_1,\ldots t_k).$$

where $\$P(t_1,\ldots t_k)$ is a sketch literal with the same arguments of the positive example and $\$P/k$ is a predicate with the same input/ output mode of p/k. ♦

4.5.1.3 Sketches as refinements

An algorithm sketch can also be seen as an internal representation of a clause that is being built according to a strategy of argument linking. The search for an adequate operational sketch is done in a space of algorithm sketches starting from an initial sketch and by employing a specific refinement operator. In that perspective, each clause is obtained by transforming an operational sketch which explains a given positive example.

Definition 4.7: An algorithm sketch is an *operational sketch* if it has no sketch literals. ♦

Definition 4.8: The process of replacing the sketch literals of a sketch by operational literals so that an operational sketch is obtained is called *sketch consolidation* . ♦

The program synthesis methodology described in this Chapter follows a strategy of sketch consolidation. When a sketch is fully consolidated, each term and each literal in the sketch are operationally linked.

Definition 4.9: A term *t* is *operationally linked* in a sketch $H \leftarrow \beta$ if and only if there is a relational link $\alpha \subseteq \beta$ from the input arguments of *H* to *t* and α contains operational literals only. ♦

Definition 4.10: A literal *L* is *operationally linked* in a sketch *Sk* if and only if all the input arguments of *L* are operationally linked in *Sk*. ♦

Although a sketch is represented as a clause, and can therefore be viewed as a set of literals, we will define the sketch consolidation algorithms assuming a given ordering of the literals in the body if the sketch. This will be done only for the sake of clarity.

Definition 4.11: A sketch $H \leftarrow \alpha$ is a *syntactically ordered sketch* if and only if the following conditions are true:

1) Every operationally linked literal appears to the left of any non-operationally linked literal.

2) Every operationally linked operational literal appears to the left of any sketch literal. ♦

Example 4.4: The sketch

r v(+[3,2,1],–[1,2,3])←
$P1(+[3,2,1],–3,–[2,1]),rv(+[2,1],–[1,2]),$P2(+3,+[1,2],-[1,2,3]).

is syntactically ordered. The sketch

r v(+[3,2,1],–[1,2,3])←
rv(+[2,1],–[1,2]),$P1(+[3,2,1],–3,–[2,1]), $P2(+3,+[1,2],-[1,2,3]).

is not ordered. Literal *$P1(+[3,2,1],–3,–[2,1])* is operationally linked, even though it is not an operational literal. Therefore it should be to the left of literal *rv(+[2,1],–[1,2])* that is not operationally linked.

The sketch

r v(+[3,2,1],–[1,2,3])←
$P2(+3,+[1,2],-[1,2,3]),$P1(+[3,2,1],–3,–[2,1]),rv(+[2,1],–[1,2]).

is not ordered either. Literal *$P2(+3,+[1,2],-[1,2,3])* is not operationally linked (none of its input terms is directionally linked) and appears to the left of *$P1(+[3,2,1],–3,–[2,1])*.

♦

4.5.2 Clause structure grammars

Another important source of information for our program synthesis methodology, and which is not part of the specification, is the *clause structure grammar*. The clause structure grammar contains programming knowledge and, for that reason, is not specific to the synthesis task of any particular predicate. Instead it is generic for a certain class of programs. One particular clause structure grammar can be used to synthesize divide-and-conquer programs, while another can describe generate-and-test programs. In our methodology, clause structure grammars are described using the definite clause grammar (DCG) notation [88]. CSG are described in Section 4.7.5.

Algorithm sketches, as well as clause structure grammars, make the synthesis task easier to accomplish. Obviously, the user has to take some time giving this information to the system. However, the clause grammars are potentially reusable (as shown) and not particular to a given program.

4.6 Class of synthesizable programs

The programs synthesized by our methodology consist of clauses with one literal in the head and without negated literals in the body. In other words, they are definite programs.

The produced logic programs do not have functors nor constants. The arguments of the literals in the clauses are always uninstantiated variables. The need for functors is

eliminated by using appropriate predicates. The process of transforming a program which contains function symbols into an equivalent one without function symbols is called *flattening* [102]. For example, the sequence of literals *p([A|B]),q(B)* that contains a structured term (the list *[A|B]*) can be represented by *p(X),decomp(X,Y,Z),q(Z)*. The predicate *decomp/3* decomposes a list *X* in head *Y* and tail *Z*. As we will see later, various auxiliary predicates similar to *decomp/3* will be used to aid the in the synthesis task. The definitions of these auxiliary predicates are added to the background knowledge and supplied to the system.

Constants are handled in a similar way. Predicates such as *null/1* or *zero/1* can be used to introduce into the clauses the constants *[]* (the empty list) and *0* (number zero), respectively.

The choice of a functor free language is not fundamental, in the sense that the methodology could be adapted to work with functors. However, the approach chosen has the advantage that it simplifies the clause refinement operation, and, consequently, the synthesis algorithms.

Nevertheless, flattened clauses produced by SKILit can be automatically unflattened by the system for presentation. Some of the synthesized programs shown here are presented in their unflattened form.

4.7 The synthesis of a logic program

The synthesis methodology employed by system SKIL takes as input a set of positive examples E^+, negative examples E^-, integrity constraints *IC* on a predicate *p/k*, a (possibly empty) initial program P_0 and background knowledge *BK*. The output is a logic program *P* that defines predicate *p/k*. The system uses a covering strategy which works as follows.

For each uncovered positive example $e \in E^+$, SKIL tries to construct a new clause, so that when added to P, e gets covered (see Algorithm 1). Clause construction is done in procedure *ClauseConstruction* (Algorithm 2). When this procedure fails to construct a new clause, the empty set ($\varnothing$) is returned. In this case, program P remains therefore unchanged, and Algorithm 1 moves on to the next positive example.

Procedure SKIL
input: E^+,E^-,IC, P_0, BK
output: P
$P := P_0$
for each $e \in E^+$ **where** $P \cup BK \cup E^+\text{-}\{e\} \nvdash e$
    $NewClause := ClauseConstruction\ (e,E^+\text{-}\{e\},E^-,IC,P,BK)$
    $P := P \cup NewClause$
next
return P

Algorithm 1: Construction of a program by SKIL

Program P can be initially empty, but may already contain some clauses which define predicate *p/k*. These initial clauses can be supplied by the user, or by another procedure invoking SKIL, as is the case of algorithm SKILit presented in Chapter 5. The initial program is P_0. Algorithm 1 shows the details of the covering procedure.

4.7.1 The clause constructor

Each clause is constructed to cover a particular positive example of the predicate to be synthesized. That example serves as a *seed* in the construction process, since its arguments are used to guide the selection of literals in the clause body.

The clause construction strategy is based on the search of a *relational link* between the input arguments of the example and the output arguments. This link is made by the admissible auxiliary predicates. In the case of recursive programs, the predicate to synthesize is itself an admissible predicate. The predicate being synthesized is partially defined by the positive examples E^+ and possibly by existing clauses (for example in P_0).

When the procedure *ClauseConstruction* is invoked by Algorithm 1 the example *e* to be covered and the set of remaining positive examples E^+-{*e*} are passed as separate arguments. Using the examples in E^+-{*e*} in the process of clause construction enables SKIL to induce recursive clauses.

The clause returned is extracted from the relational link, i.e., from the sequence of literals that link the input arguments of the positive example to its output arguments. This last step involves mainly transforming constants into variables.

Example 4.5: Suppose we have the following scenario:

Positive example (with mode declaration):

mode(grandfather(+,-)).
grandfather(tom,bob).

Background knowledge (with mode declarations):

mode(father(+,-)).	*mode(mother(+,-)).*
father(tom,anne).	*mother(anne,bob).*
father(tom,jack).	*mother(anne,chris).*

To construct a clause that covers the given positive example we will try to link the input argument (*tom*) with the output argument (*bob*). The sequence of literals *father(tom,anne),mother(anne,bob)* establishes that link and can be regarded as a sort of explanation of the positive example.

Now, with the positive example and that sequence of literals we will construct the instantiated candidate clause (sketch)

grandfather(tom,bob)←father(tom,anne),mother(anne,bob)

from which we extract the clause

grandfather(X,Z)←father(X,Y),mother(Y,Z)

by replacing constants with variables. ♦

The process of constructing a clause that, together with the background knowledge, covers a given positive example, consists mainly in the consolidation of a sketch associated to that example. The sketch associated to an example *e* is either supplied by the user or a black box sketch of the form *e*←*$P(…)* which is automatically generated by the system (see Section 4.5.1).

The consolidation of a sketch is done using a *breadth-first search* strategy. The objective is to obtain what is called an *operational sketch*, i.e., a sketch without sketch literals. The search is conducted using a *refinement operator* ρ, which provides the set of refinements of every sketch. A sketch refinement is also a sketch. The search starts with the sketch associated to the positive example.

Procedure ClauseConstruction
input: e, E^+, E^-, IC, P, BK
output: *Cl* (the new clause)
Sketch := *AssociatedSketch(e)*
Q := [*Sketch*]
repeat
  if $Q = \emptyset$ **then return** $\emptyset$
  Sk := first sketch in *Q*
  if *Sk* is an operational sketch **then**
    Cl := *Variabilize(Sk)*
    if $\{Cl\} \cup P \cup BK \cup E^+$ covers *e* **and**
      $\{Cl\} \cup P \cup BK \cup E^+$ does not cover any $e \in E^-$ **and**
      $\{Cl\} \cup P \cup BK \cup E^+$ does not violate *IC*
      then return {*Cl*}
    end if
  end if
  Q := *Q* - *Sk*
  NewSk := $\rho(Sk, P, BK, E^+)$ (Algorithm 3)
  Q := *Q* after appending *NewSk* to the end of *Q*
always

Algorithm 2: Generation of a clause through the refinement of a sketch

The search stops when an operational sketch is found which satisfies the stopping criterion. The clause returned is obtained by replacing the sketch terms with variables (a

process we call *variabilization*). Variabilization of the clause is done by function *Variabilize* described in Section 4.7.1.1.

The procedure *ClauseConstruction* (Algorithm 2) initializes a queue *Q* of refinements with the sketch associated to the example given as input. In every iteration of the 'repeat' cycle, the first sketch in *Q* is removed, and a set of its refinements is constructed. The sketches in the refinement set are placed at the end of *Q*.

As we can see, the repeat cycle may terminate for different reasons. Ideally, it stops when an operational sketch is found. From that sketch is extracted a clause that covers the positive example and does not violate the integrity constraints or cover any negative example. In order not to violate the negative examples, $\{Cl\} \cup P \cup BK \cup E^+$ cannot intensionally cover any of them. Integrity constraints are checked by the module MONIC, described in Chapter 7. When the refinement queue *Q* becomes empty Algorithm 2 stops as well. In this case, the empty set is returned.

In the current implementation of the SKIL system, the number of refinements constructed during the generation of a clause is also controlled. For that, we impose a limit on the number of refinements constructed. This parameter is called *effort limit*. Its default value is of 300 refinements, but it can be set using a specific declaration. When the effort limit is reached, the construction of the clause terminates, and the empty set is returned.

4.7.1.1 Variabilization

The *variabilization* of a sketch consists of replacing the terms occurring in the sketch by variables. This replacement can be done using different variabilization strategies. Here we describe two of them: the *simple variabilization strategy* and the *complete variabilization strategy*.

To variabilize a sketch using the simple variabilization strategy, each term is replaced with a variable. The same variable corresponds to different occurrences of the same term.

For example, the clause extracted from the sketch *p(a,z)←q(a,c),t(a,c,z)* is *p(A,Z)←q(A,C),t(A,C,Z)*. This is the simplest variabilization method which assumes that two different variables correspond to two different terms. Under this assumption the variabilization of a sketch is unique.

The complete variabilization procedure returns, for each sketch, the set of clauses that have that sketch as an instance. The complete variabilization of sketch *p(a,z)←q(a,c),t(a,c,z)* is a set of 20 clauses including

p(A,Z)←q(A,C),t(A,C,Z),
p(A,Z)←q(B,C),t(A,C,Z),
p(A,Z)←q(A,C),t(B,C,Z),
p(A,Z)←q(A,C),t(A,D,Z),

etc.

If the function *Variabilize* uses the complete variabilization procedure then it returns a set of clauses instead of just one. In this case the stopping conditions of Algorithm 2 must be checked for each clause resulting from the variabilization. The algorithm stops if one of the clauses satisifies the conditions. The result of *ClauseConstruction* is then the set of variabilizations (clauses) satisfying the stopping criterion.

In the current implementation of SKIL only the simple variabilization procedure is available. The variabilization strategy can, however, be an option of the user. Other variabilization strategies could also be devised.

4.7.2 The refinement operator

The set of refinements of a sketch *Sk* is given by the refinement operator ρ (Algorithm 3). This operator takes *Sk* and selects one sketch literal *$P(X,Y)* to consolidate (*X* represents the set of input arguments and *Y* the output). The job of the refinement operator is to find all possible replacements for this sketch literal. Each replacement is

made of an operational literal and a new sketch literal. Ultimately, the sketch literal *$P(X,Y)* can also be removed.

The refinement operator always consolidates the sketch from input to output, i.e., it only introduces operational literals whose input arguments are linked to the input arguments of the head of the sketch via operational literals only. Therefore, the selected *$P(X,Y)* must be a literal whose input arguments *X* are operationally linked terms within the sketch. If more than one such sketch literal exists, the leftmost one is chosen for refinement. To simpify the description of Algorithm 3 we assume that the sketch to refine is *syntactically ordered* (Section 4.5.1). This means that the selected *$P(X,Y)* is always the leftmost sketch literal.

Procedure ρ
input: algorithm sketch *Sk*
P,BK,E^+
output: a set of sketch refinements of *Sk*
Sk := $e \leftarrow \alpha, \$P(X,Y), \beta$ where
α and β are literal sequences,
$P(X,Y) is the leftmost sketch literal whose
input arguments are directionally linked terms.
X is the set of its input arguments,
Y is the set of its output arguments,
if there is no *$P(X,Y)* in those conditions **return** $\emptyset$
RelMod := *RelevantSubModel(X,P,BK,*E^+*,*$e \leftarrow \alpha$*)*
NewLiterals := {(*Pred(*X_M*,*Y_M*),$Pnew(*$X \cup Y_M$*,* $Y - Y_M$*))* |
Pred(X_M*,*Y_M*)*$\in$ *RelMod* and *$Pnew* is a new sketch literal }
Refin := { $e \leftarrow \alpha, \gamma, \beta$ | $\gamma \in$ *NewLiterals* }
if *Y*=$\emptyset$ **then** *Refin* := *Refin* $\cup$ { $e \leftarrow \alpha, \beta$ }
return *Refin*

Algorithm 3: Refinement Operator

Having identified the sketch literal *$P(X,Y)* to refine, the method constructs a set of atoms that belong to the model of $P \cup BK \cup E^+$. Each of these atoms has as input arguments terms in *X*. This set of atoms is the *relevant sub-model* (see Algorithm 4).

Each element $Pred(X_M,Y_M)$ of the relevant sub-model *ModRel* will correspond to one refinement. For that, the sketch literal is replaced by a conjunction $Pred(X_M,Y_M)$, $\$Pnew(X_{Pnew},Y_{Pnew})$, where *$Pnew* is the new sketch predicate. The new sketch literal represents new consolidation opportunities in subsequent refinement steps. The set of input terms X_{Pnew} includes the terms in X and in Y_M. The set of output terms Y_{Pnew} includes the terms in Y that are not operationally linked yet.

If the set of output terms Y in *$P(X,Y)* is empty the refinement obtained by simply removing this sketch literal is also returned. Making one sketch literal disappear allows SKIL to move on to the next sketch literal and eventually consolidate the whole sketch.

Example 4.6: Let *Sk* be the sketch

grandfather(+tom,–bob)←father(+tom,–anne),$P1(+tom,+anne,–bob).

Sk has one sketch literal (*$P1(+tom,+anne,–bob)*). Each element of the set of refinements is constructed by replacing this sketch literal by a conjunction of an operational literal and of a new sketch literal. Here is the refinement set, using the predicates defined in Example 4.5:

Refin = { (*grandfather(+tom,–bob)←*
father(+tom,–anne),
mother(+anne,–bob),
$P2(+tom,+anne,+bob).),
(*grandfather(+tom,–bob)←*
father(+tom,–anne),
mother(+anne,–chris),
$P3(+tom,+anne,+chris,–bob).) }

♦

4.7.3 The relevant sub-model

The operational literals that replace the sketch literal, correspond to a set *RelMod* of ground facts derived from the program $P \cup BK \cup E^+$. This *RelMod* set is a relevant subset of the model of $P \cup BK \cup E^+$ (which we call the relevant sub-model) and is constructed as

follows (see Algorithm 4). For each admissible predicate we construct queries using input arguments of the sketch literal. The queries are posed to the program $P \cup BK \cup E^+$ using a depth-bounded program interpreter (Section 4.7.4). The set of answers given by the interpreter is the intended sub-model *RelMod*.

Procedure *RelevantSubModel*
input: X, P,BK,E^+,$e \leftarrow \alpha$
output: *RelMod* a relevant sub-model of $P \cup BK \cup E^+$
RelMod := $\varnothing$
Predicates := *PredicatesToFollow*($e \leftarrow \alpha$)
for each *Pred* $\in$ *Predicates*
    Queries := { $Pred(X_p,Y_p) \mid X_p \subseteq X$, Y_p are variables }
    Atoms := { $Q\theta \mid Q \in Queries$ and $\theta \in Int(P \cup BK \cup E^+, Q, \vdash)$ }
    RelMod := *RelMod* $\cup$ *Atoms*
next
RelMod := *RelMod* $-\alpha$ (eliminates literal repetitions)
RelMod := *Prune*(*RelMod*, $e \leftarrow \alpha$)
return *RelMod*

Algorithm 4: Construction of the relevant sub-model

Example 4.7: The input arguments {*tom, anne*} of the sketch literal *$P1(+tom, +anne, -bob)* in the following sketch

grandfather(+tom,-bob)$\leftarrow$
    father(+tom,-anne),
    $P1(+tom,+anne,-bob).

are used to formulate queries involving the admissible predicates *father/2* and *mother/2* (assuming that these are admissible predicates). Taking the definitions for these predicates given for Example 4.5, we get the following set of possible queries

Queries = { *father(tom,X), father(anne,X), mother(tom,X), mother(anne,X)* }

The first and fourth query get two answer substitutions each. The second and third queries get no answers. The set of facts constructed from the answers is

Facts = { *father(tom,anne), father(tom,jack),*

mother(anne,bob), mother(anne,chris) }

The relevant sub-model is

RelMod = {*father(tom,jack), mother(anne,bob), mother(anne,chris)*}

It should be stressed that *father(tom,anne)* was excluded from the relevant sub-model as it is already in the sketch being refined.♦

Why are we interested in a sub-model of $P \cup BK \cup E^+$? The background knowledge *BK* enables the introduction of auxiliary predicates. The positive examples E^+ enable the introduction of recursive literals. The previously induced clauses in *P* speed up the induction of recursive clauses. Although we can learn recursive clauses from relevant sub-models of $BK \cup E^+$ only (without *P*), this would make the success of the system very much dependent on the choice of the positive examples. This issue will be elaborated in the following Chapter.

Algorithm 4 removes from the relevant sub-model atoms that already exist as literals in the sketch that is being refined. This control avoids the unnecessary repetition of literals in the final clause.

4.7.3.1 Pruning

The function *Prune* is made of two different heuristic steps described below. A non-heuristic version of Algorithm 4 can be obtained by removing the call to the function *Prune*.

First heuristic step:

RelMod := *RelMod*–{*e*' | *e*' has the same predicate as *e*
and its input arguments are
a subset of the input arguments of *e*}

Second heuristic step:

RelMod := *RelMod*–{ *L* | *L* introduces terms produced by $e \leftarrow \alpha$ }

In the first heuristic step, atoms corresponding to recursive literals that are potential sources for non-termination are removed. The criterion is that all the atoms whose input arguments are a subset of the input arguments of the head of the sketch are removed. Thus, we will not have such clauses as $p(X) \leftarrow p(X)$ nor as $p(X,Y) \leftarrow p(Y,X)$. This is an elementary control of non-termination, which does not avoid all undesirable situations. In any case, the program interpreter used in SKIL has itself a mechanism to prevent non-termination: the control of the depth of demonstrations.

The second heuristic step removes from the relevant sub-model atoms that try to re-introduce terms already existing in $e \leftarrow \alpha$. The set of output terms of an atom L in the relevant sub-model must be disjoint from the *set of produced terms* in $e \leftarrow \alpha$.

Definition 4.12: Given a clause $e \leftarrow \alpha$, and the input/output mode declarations of the predicates involved, the set of terms *produced* by the clause is

$$in(e) \cup \{ \text{ directionally linked terms of } \alpha \text{ with respect to } in(e) \}$$

where *in(e)* is the set of input terms of the head of the clause. The set of terms produced by $e \leftarrow \alpha$ is denoted by $produced(e \leftarrow \alpha)$. ♦

So, any atom L of the relevant sub-model generated by Algorithm 4 must satisfy the following condition:

$$out(L) \cap produced(e \leftarrow \alpha) = \varnothing$$

where *out(L)* denotes the set of output terms of atom L.

Atoms not satisfying this restriction are discarded because, after variabilization of the sketch, they would correspond to potentially useless literals. This is a reasonable heuristic since the aim of the refinement process is to produce the output terms of the example, and it is typically unnecessary to produce each term more than once. However,

under this heuristic and given one example, some clauses covering it may not be synthesizable.

Example 4.8: Let $e \leftarrow \alpha$ in Algorithm 4 be *rv(+[3,2],–[2,3])←dest(+[3,2],–3,–[2])*. In this case the atom *rv(+[2],–[2])* will not be in *RelMod* because

out(rv(+[2],–[2])) = {*[2]*}
produced(rv(+[3,2],–[2,3])←dest(+[3,2],–3,–[2])) = { *[3,2], 3, [2]* }
{*[2]*} ∩ { *[3,2], 3, [2]* } = {*[2]*} ≠ ∅

Therefore the clause *rv(A,B)←dest(A,C,D), rv(D,D), const(B,C,D)* is never synthesized. ♦

The use of this filter reduces the number of possible sketch refinements at each refinement step, as well as the branching factor of the search process thus increasing efficiency.

However, this filter has the disadvantage of causing incompleteness in the clause construction.

Example 4.9: Suppose that example e_1 is *rv([1,2],[2,1])*. The recursive clause is

rv(A,B)←dest(A,C,D),rv(D,E),addlast(E,C,B).

The sketch that SKIL should find is

rv([1,2],[2,1])←dest([1,2],1,[2]),rv([2],[2]),addlast([2],1,[2,1]).

This sketch is never produced by SKIL from example *rv([1,2],[2,1])*. When SKIL refines *rv([1,2],[2,1])←dest([1,2],1,[2]),$Px(…)*, the atom *rv([2],[2])* is not allowed into the relevant sub-model because it attempts to re-introduce the term *[2]*.♦

4.7.4 The depth bounded interpreter

SKIL's synthesis methodology employs SLD/SLDNF resolution in the following situations:

- Tests for the coverage of positive and negative examples;
- Construction of the relevant sub-model.

The SLD-resolution may give rise to practical problems due to the possibility of having infinite or too long computations. To guarantee the termination of the synthesis process, the program interpreter used by SKIL employs a mechanism that controls the depth of each refutation.

Definition 4.13: Let D be a derivation of a program P. The *invocation level* of an occurrence C_i of a clause $C \in P$ in a derivation D is defined as $invl(C_i,D)$:

$invl(C_i,D) = 0$ if C_i is in the first step of the derivation, i.e., $D = ((Q,C_i,\theta),\ldots)$.

$invl(C_i,D) = k+1$ if C_i resolves with a literal first appearing in resolvent R_{j+1} in D, R_{j+1} is obtained by resolving R_j and C_j, and $invl(C_j,D)=k$. ♦

Example 4.10: Consider the following zero-order program:

$a \leftarrow b,a.$	C_1
$a \leftarrow c.$	C_2
$b.$	C_3
$c.$	C_4

One possible derivation is shown in Figure 4.7.

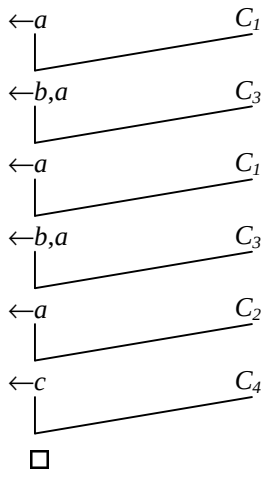

Figure 4.7: One derivation of the program.

Symbolically, the derivation is represented by

$$D = ((\leftarrow a, C_{1,1}), (\leftarrow b, a, C_{3,1}), (\leftarrow a, C_{1,2}), (\leftarrow b, a, C_{3,2}), (\leftarrow a, C_{2,1}), (\leftarrow c, C_{4,1}), \square)$$

(substitutions are not considered since they are not needed) where $C_{k,i}$ represents the i-th occurrence of clause C_k.

The invocation level of $C_{1,1}$ is 0 since it is in the first step of the derivation. The invocation level of $C_{3,1}$ is 1

$$invl(C_{3,1}, D) = 1 + invl(C_{1,1}, D)$$

since $C_{3,1}$ resolves with literal b introduced by $C_{1,1}$. The invocation level of $C_{1,2}$ is also 1. As for the rest of the derivation,

$$invl(C_{3,2}, D) = 2 = 1 + invl(C_{1,2}, D) = 1 + 1$$

$$invl(C_{2,1}, D) = 2 = 1 + invl(C_{1,2}, D) = 1 + 1$$

$$invl(C_{4,1}, D) = 3 = 1 + invl(C_{2,1}, D) = 1 + 2$$

♦

Now we can define the depth of a refutation in terms of the maximum invocation level of a clause on all the derivations of an SLD tree.

Definition 4.14: Let P be a definite logic program and $\leftarrow Q$ a query. The *refutation depth*, *refdepth*($\leftarrow Q,P$), of $\leftarrow Q$ from P is the maximum invocation level of all clause occurrences in the SLD derivation tree T of $\leftarrow Q$:

$$refdepth(\leftarrow Q,P) = max(\{\ invl(C_i,D) \mid D \text{ is a branch of } T \text{ and } C_i \text{ occurs in } D\ \})$$

♦

The two above notions can be extended to SLDNF resolution in a natural way.

The depth-bounded interpreter answers only those queries which admit a refutation with depth smaller then a given limit h. When the depth of a demonstration goes beyond the limit, the interpreter fails.

Definition 4.15: Let P be a program and $\leftarrow Q$ a query, a *depth-bounded interpreter* of limit h is the operator,

$$Int(P,\leftarrow Q, \vdash_h) = \{\ \theta \mid P \vdash_h Q\theta\ \}$$

where $\vdash_h$ represents the derivability relation

$P \vdash_h Q$ if and only if $P \vdash Q$ and $refdepth(\leftarrow Q,P) \leq h$.

♦

In ILP approaches it is common to find some sort of control of the computation depth. The interpreter used in SKIL employs a control mechanism similar the one used by Shapiro for MIS [109] to diagnose cyclic programs. Muggleton and Feng used the notion of h-easy model to construct subsets of a program model [82].

Definition 4.16: Given a logic program *P*, an atom *q* is *h-easy* with respect to *P* if and only if there is a derivation of *q* from *P* involving at most *h* resolution steps. The Herbrand *h-easy model* of *P* is the set of all the instances of atoms h-easy with respect to *P*.

The h-easy model of a program *P* corresponds, in broad terms, to the set of facts which can be derived with a depth-bounded interpreter. To guarantee that the h-easy model is finite, program clauses should be range restricted.

The h-easy approach was criticized by de Raedt who, instead, proposed to limit the complexity of the terms involved in each computation [21].

Definition 4.17: An atom $f(t_1,\ldots,t_n)$ is *h-complex* if and only if for all i: $1 \leq i \leq n$: depth of term $t_i \leq h$ (page 47). ♦

An *h-complex model* of a program *P* corresponds to the set of atoms which have a derivation from *P* involving only h-complex terms. A program *P* is *h-conform* if, for every h-complex atom *q*, the SLD tree for deriving *q* from *P* only contains h-complex atoms.

Although it seems simple to adopt the h-complex approach for controlling termination in SKIL, we believe that the practical results obtained by the synthesis method would not be much different if the h-complex approach was adopted. On the other hand, a complexity-bound interpreter would be computationally heavier. For h-conform programs the control of complexity could be done statically. Unfortunately, for a program to be h-conform, severe syntactic restrictions must be imposed. One of those conditions is that all variables occurring in the body of a clause also occur in the head. This is not adequate for our purposes.

4.7.5 Vocabulary and clause structure grammar (CSG)

The admissible predicates that can be used to obtain the sub-model are given by the function *PredicatesToFollow* invoked by Algorithm 4. Those predicates are determined, beforehand, by the admissible predicates declaration. They constitute the *vocabulary* available for clause construction. The function *PredicatesToFollow* can be defined in a simple form, returning the set of vocabulary predicates. This is the solution usually adopted by ILP systems.

However, it is sensible that the semi-automatic development of programs should explore *programming knowledge* [38,111]. The knowledge relative to the processing of structured objects such as lists could include, for example, the following. If we want to process an object using a procedure *P*, we decompose that object into parts, invoke the same procedure recursively, and combine the partial solutions. The SKIL system allows this kind of programming knowledge to be expressed as a *clause structure grammar* (CSG).

A clause structure grammar defines the admissible sequences of predicate names in the body of synthesized clauses. Such CSG's are expressed in *definite clause grammar* (DCG) notation [88].

The top rules of the CSG's used here have the form

$$body(P) \text{--}> L_1(<O_1>,<N_1>),\ldots,recurs(<O_r>,<N_r>,P),\ldots,L_n(<O_n>,<N_n>).$$

where for each $L_i(<O_i>,<N_i>)$

- L_i is the name of a group of literals (e.g. test literals, decomposition literals, etc.),
- $<O_i>$ is either * or +. The symbol * means that the sequence of literals can be empty. The symbol + means that there should be at least one literal in the sequence.
- $<N_i>$ is an integer greater than 0, which limits the maximum admissible number of literals in the group.

- *P* is a DCG variable.

The group *recurs* is a special group for recursive literals. The only predicate admissible in this group is the predicate being synthesized. Its name is carried in variable *P*.

For each L_i the CSG contains a set of rules of the form

L_i(_,N)-->lit_L_i,{N>0}.
L_i(_,N)-->lit_L_i,{N2 is N-1},L_i(+,N2).
L_i(,N)-->[].*

lit_L_i-->[<P_1>];...;[<P_k>].

where each <P_i> is a predicate of the group L_i, *lit_L_i* is a DCG predicate name, and *N*, *N2* are DCG variables.

The *recurs* special a group is defined with the set of rules

recurs(_,N,P)-->lit_recurs(P),{N>0}.
recurs(_,N,P)-->lit_recurs(P),{N2 is N-1},recurs(+,N2,P).
recurs(,N,P)-->[].*

lit_recurs(P)-->[P].

Example 4.11: The CSG shown here describes a set of recursive clauses. It starts by defining several groups of literals. The first group decomposes certain arguments of the clause head in sub-terms (using predicates like *dest/3*, which separate a list into head and body). The second group contains test literals. The third group allows the introduction of recursive literals. Finally, the fourth group consists of composition literals, whose purpose is to construct the output arguments from terms obtained by previous literals (using predicates like *append/3*). The general structure of the recursive clause is described in the following way:

body(P)-->decomp(+,2),test(,2),recurs(*,2,P),comp(*,2).*

where the argument *P* passes the name of the predicate in the head (for example *member/2* if we are synthesizing *member*). The maximum number of literals of any given group is 2. All the groups of literals may be empty except for the *decomp* group. The decomposition group is defined following the model defined above:

decomp(_,N)-->lit_decomp,{N>0}.
decomp(_,N)-->lit_decomp,{N2 is N-1},decomp(+,N2).
decomp(,N)-->[].*

lit_decomp-->[dest/3];[pred/2];[partb/4].

The group of recursive literals is also defined as above. The test and composition groups are defined similarly to the decomposition group. Below we show only the *lit_test* and *lit_comp* rules.

lit_test-->[null/1];[memberb/2].
lit_comp-->[appendb/3]; [addlast/3];[const/3].

Some clauses admited by this CSG (assuming in this example that we are synthesizing *rv/2*) would have the form

rv(_,_)←dest(_,_,_),rv(_,_).
rv(_,_)←pred(_,_,_),rv(_,_).
rv(_,_)←dest(_,_,_),rv(_,_),addlast(_,_,_).

Some clauses not admited by the CSG:

rv(_,_)←rv(_,_).

Clauses must have at least one decomposition literal.

rv(_,_)←rv(_,_),dest(_,_,_),rv(_,_).

No clause can have a decomposition literal between two recursive literals.

rv(_,_)←dest(_,_,_),dest(_,_,_),dest(_,_,_).

The maximum number of decomposition literals is 2.♦

When Algorithm 4 invokes the function *PredicatesToFollow,* with the part of the sketch $e \leftarrow \alpha$ to the left of the literal *$P(…)* as an argument, it generates the set of admissible predicate names which, according to the CSG, can follow α. The CSG does not restrict the literal arguments. It simply defines acceptable predicate chains that can appear in the literals of the body of a clause.

It would be relatively simple to extend the CSG to restrict the arguments of the literals also. However, we prefer to adopt this simple solution since it makes CSGs easier to write and maintain. In any case, the choice of literal arguments is restricted by the clause construction mechanism that always follows some relational link and takes the types of the predicates into account.

The function *PredicatesToFollow* invokes the predicate *body/3* defined by the CSG in the following way: the first argument is instantiated to the name of the predicate to be defined; the second argument is a list whose first elements represent the sequence of the predicate names in α. The next element of that list is a variable, which will be instantiated with the predicate name that can follow in the sequence. The rest of the list is a non-instantiated variable. The third argument is an empty list.

Example 4.12: $e \leftarrow \alpha$ is the clause

sort([2,1],[1,2])←dest([2,1],2,[1]).

Thus, given the CSG from Example 4.11, the set of predicates that can follow is {*dest/3, partb/4, sort/2*}. This is equivalent to collecting the answers obtained by the query.

←body(sort/2,[dest/3,PRED|_],[]).

Variable *PRED* will be successively unified with *dest/3*, *partition/4* and *sort/2*. If we considered all the vocabulary predicates, independently of CSG, then the set of *PredicatesToFollow* would be

{*dest/3, partb/4, null/1, memberb/2, sort/2,const/3, appendb/3,addlast/3*}

♦

Clause structure grammars enable the description of an adequate language bias. The method is quite powerful since each grammar can be highly reusable. The same grammar can cover a large class of predicate definitions.

4.7.6 Type checking

The types declared in the specification are also checked during the construction of the relevant sub-model. This step was not explicitly included in Algorithm 4 for the sake of clarity. In reality, the set of queries constructed by the instruction

$$Queries := \{\ Pred(X_p,Y_p) \mid X_p \subseteq X,\ Y_p \text{ are variables } \}$$

of Algorithm 4 excludes those queries whose input arguments do not conform to the type declaration. For that, SKIL checks if every input term is in the domain of the corresponding type. In other words, the system checks whether the n-tuple of the query arguments is *compatible* with the type declarations (Section 3.2.4). This checking is made using the type definitions (see Appendix B). For the predicates whose type is not declared any input terms are accepted.

4.8 Properties of the refinement operator

In this Section we discuss some theoretical properties of SKIL's refinement operator. We are mainly interested in determining if the refinement operator can always find a clause covering a given example if the clause is in the search space.

Given a program P and an example $e(X,Y)$ such that $in(e(X,Y))=X$ and $out(e(X,Y))=Y$, if there is a relational link α from X to Y such that $P \vdash \alpha$, then SKIL's refinement operator (ρ) finds it.

If we have a positive example with no sketch associated, the refinement operator ρ starts with the black box sketch *e(X,Y)←$P1(X,Y)* and finds all the refinements *e(X,Y)←p(X2,Y2),$P2(X3,Y3)* such that $P \vdash p(X2,Y2)$ and $X2 \subseteq X$, where *$P2(X3,Y3)* is a new sketch literal whose arguments *(X3,Y3)* are a combination of *(X,Y)* and *(X2,Y2)*. The repeated application of ρ gives all the relational links from *X* to *Y*. If there is one sketch associated, the refinement operator handles each sketch literal in a similar way. Given a program *P* and a sketch *Sk* such that, if there is a clause *C* that is a variabilization of a consolidation of *Sk* then SKIL can find that clause.

Now we give a formal account of what has been stated above. We show that SKIL's refinement operator can find all the interesting operational refinements of a given sketch. As a consequence SKIL can find all the variabilizations of those refinements. We start by defining the concept of consolidation. The interesting refinements of a sketch will be its consolidations.

Note that in the following we use a clause-like notation for representing sets of literals. The sequence α_1,α_2 represents the set of literals $\alpha_1 \cup \alpha_2$, where α_1 and α_2 are sets of literals. The sequence L,α represents the set $\{L\} \cup \alpha$, where *L* is a single literal and α is a set of literals.

Definition 4.18: A set of literals α is a *consolidation* of a set β of operational or sketch literals, denoted $\alpha \angle \beta$ iff:

a) $\alpha = \beta$;
b) β is of the form *$P(X,Y)* and α is a relational link from the set of terms $SX \subseteq X$ to the set of terms $SY \supseteq Y$;
c) β is of the form (L,β_2), where *L* is an operational or sketch literal, α is of the form (α_1,α_2), $\alpha_1 \angle L$ and $\alpha_2 \angle \beta_2$. ♦

Intuitively, a set of literals is a consolidation of a sketch literal *$P(X,Y)* if it produces all the output terms *Y* of *$P(X,Y)* from a subset of its input terms *X*. Note that the empty set

is an acceptable consolidation for any sketch literal with no output terms. The notion of consolidation is recursively extended to arbitrary sets of literals.

Example 4.13: Suppose we have two predicates *p(+,–)* and *q(+,–,–)*. The set of literals *p(a,b),q(b,c,d)* is one possible consolidation of the sketch literal *$P1(+a,–c,–d)* since there is a relational link from {*a*} to {*c,d*}. We also have that *p(+a,–b),q(+b,–c,–d)* is one consolidation of *$P2(+a,–c)* since, in particular, there is a relational link from {*a*} to {*c*}. The empty set is one consolidation of *$P3(+a,+b)*. Another consolidation of this sketch literal is *p(+a,–b),p(+b,–c)*.

One consolidation of *p(+a,–b),$P4(+b,-d),p(+d,–f)* is *p(+a,–b),p(+b,–c),$P5(+b,+c,–d),p(+d,–f)*.♦

One sketch is a consolidation of another sketch if both have the same head and there is a relation of consolidation between their bodies.

Definition 4.19: Let S_1 and S_2 be two sketches. S_2 is a *consolidation* of S_1, denoted $S_2 \angle S_1$, iff $S_1=(H \leftarrow \alpha_1)$, $S_2=(H \leftarrow \alpha_2)$ and $\alpha_2 \angle \alpha_1$.♦

A sketch refinement operator produces consolidations of one sketch.

Definition 4.20: A *sketch refinement operator* (SRO) ρ is an operator that, given a sketch *S*, returns a set of sketches, denoted by $\rho(S)$, where for all $S' \in \rho(S)$ we have that $S' \angle S$.♦

SKIL's refinement operator has four arguments: $\rho(S,P_0,BK,E^+)$. The first argument is the sketch to refine. The others are the initial program P_0, the background knowledge *BK* and the positive examples E^+. In this section we consider these last three arguments as one single program $P = P_0 \cup BK \cup E^+$. For the same reason we invoke *RelevantSubModel* with the empty set in the third and fourth arguments. As shorthand for $\rho(S,P_0,BK,E^+)$ we write $\rho(S)$.

Definition 4.21: The set of refinements of a sketch S obtained by iterated application of a SRO ρ is denoted as $\rho^*(S) = \{S\} \cup \rho(S) \cup \rho^2(S) \cup \rho^3(S) \cup \ldots$. ♦

We now define the notion of completeness of a sketch refinement operator in terms of the notion of consolidation.

Definition 4.22: Let ρ be a SRO, SS a set of sketches, S_1 a syntactically ordered sketch in SS, and S_2 an operational sketch in SS such that $S_2 \angle S_1$. The SRO ρ is *complete* in SS iff $S_2 \in \rho^*(S_1)$. ♦

Theorem 4.1: Given a program P, SKIL's refinement operator, ρ, is complete in the set of sketches $SS = \{S \mid$ for every operational literal L in the body of S, $P \vdash L\ \}$.

Proof: Let S be an operational sketch in SS and S_1 an arbitrary sketch in SS such that $S \angle S_1$. We must prove that $S \in \rho^*(S_1)$,.

If S_1 has no sketch literals then, by definition of consolidation $S = S_1$. By definition of ρ^*, we have that $S \in \rho^*(S_1)$.

If S_1 has at least one sketch literal, then S_1 is of the form $H \leftarrow \alpha_1, \$P(X,Y), \beta_3$, where α_1 is a sequence of operational literals. By definition of consolidation S is of the form $H \leftarrow \alpha_1, \alpha_2, \alpha_3$, where $\alpha_2 \angle \$P(X,Y)$ and $\alpha_3 \angle \beta_3$.

If $\alpha_2 = \varnothing$ then the set of output terms Y must be empty, otherwise we would not have that $\alpha_2 \angle \$P(X,Y)$. In this case $(H \leftarrow \alpha_1, \beta_3) \in \rho(H \leftarrow \alpha_1, \$P(X,Y), \beta_3)$ since, if Y is empty, one of the refinements is obtained by eliminating the sketch literal $\$P(X,Y)$.

If $\alpha_2 \angle \$P(X,Y)$ and $\alpha_2 \neq \varnothing$ then there must be an operational literal $L \in \alpha_2$ such that $in(L) \subseteq X$. Suppose there is no such literal, then no term in Y is directionally linked in α_2 with respect to X which contradicts $\alpha_2 \angle \$P(X,Y)$.

Since $P \vdash L$ we have that $L \in$ *RelevantSubModel*$(X,P,\varnothing,\varnothing, H \leftarrow \alpha_1)$. This is justified by the fact that the relevant sub-model is obtained by constructing all queries with all allowed predicates for $H \leftarrow \alpha_1$ with all the possible combinations of input arguments taken from X. Therefore $(H \leftarrow \alpha_1, L, \$P2(X \cup in(L), Y\text{-}out(L)), \beta_3) \in \rho(S_1)$.

Now let α_2' be α_2 without L. We have that $\alpha_2' \angle \$P2(X \cup in(L), Y\text{-}out(L))$ because α_2 links $SX_2 \cup out(L) \subseteq X \cup out(L)$ to $SY_2\text{-}out(L) \supseteq Y\text{-}out(L)$. Therefore we can reason for α_2' as we did for a_2 and conclude that $H \leftarrow \alpha_1, \alpha_2, \beta_3 \in \rho^{n+1}(S_1)$ assuming that α has n literals.

Applying the same reasoning to the other sketch literals of S_1 as we did for *$P(X,Y)* we can conlude that $H \leftarrow \alpha_1, \alpha_2, \alpha_3 \in \rho^k(\ H \leftarrow \alpha_1, \$P(X,Y), \beta_3\ )$, for some integer k, i.e., $S \in \rho^*(S_1)$. ♦

If a clause structure grammar G is considered, the set of sketches SS is restricted to the sketches admited by G.

Theorem 4.2: Given a program P, a sketch S and a clause $C = H_C \leftarrow B_C$, if there is a substitution θ such that $C\theta \angle S$ and $P \vdash B_C\theta$, then SKIL can find clause C, assuming that the complete variabilization (Section 4.7.1.1) technique is used.

Proof: By the completeness of ρ and the assumption that $P \vdash B_C\theta$ we have that $C\theta \in \rho^*(S)$. Therefore SKIL can find the sketch $C\theta$ and as a consequence it can find all the variabilizations of $C\theta$ including the clause C. ♦

4.9 A session with SKIL

We start by using the SKIL system to synthesize the predicate *rv/2*. This examples helps to illustrate how the system works when well-chosen positive and negative examples are provided, and when a background knowledge program and a clause structure grammar are given. The result is a recursive program. At the end the system indicates the CPU time taken (in seconds) and the total number of sketch refinements constructed.

Specification

```
mode( rv(+,-) ).
type( rv( list,list ) ).

rv([],[]).
rv([1,2,3],[3,2,1]).
rv([2,3],[3,2]).

-rv([1,2],[1,2]).
-rv([1,2,3],[2,1,3]).
-rv([1,2,3],[2,3,1]).
-rv([1,2,3,4],[3,4,2,1]).
```

Programming knowledge

```
background_knowledge( list ).                    % Appendix A
clause_structure( decomp_test_rec_comp_2 ).      % Appendix C
adm_predicates( rv/2, [const/3,dest/3,null/1,addlast/3,rv/2] ).
```

SKIL output:

```
?- skil(rv/2).

example to cover: rv([],[])
clause c(12) generated after 2 refinements:
rv(A,A)←
   null(A).

example to cover: rv([1,2,3],[3,2,1])
clause c(13) generated after 32 refinements:
rv(A,B)←
   dest(A,C,D),
   rv(D,E),
   addlast(E,C,B).

example to cover: rv([2,3],[3,2])
example covered by existing clause c(13)
Program generated (prv):

c(12):rv(A,A)←
   null(A).
c(13):rv(A,B)←
```

```
    dest(A,C,D),
    rv(D,E),
    addlast(E,C,B).

34 refinements (total)
2.200 secs
```

The background knowledge (*list*) contains the definitions and declarations of type and mode of auxiliary predicates (Appendix A). The clause structure grammar uses a divide-and-conquer strategy as the one shown in Example 4.11 (Appendix C). Each recursive clause has in the body a sequence of decomposition literals, test literals, recursive literals, and composition literals.

By running SKIL with the same data, but without using a CSG, we obtain the same program. Nevertheless, the number of refinements increases to 60 (almost doubles) in a relatively simple problem. The processing time is also higher (about 2.7 seconds).

Type declarations also affect the system performance. We experimented removing only the type declaration for the auxiliary predicate *addlast/3*. The number of refinements was 84 (instead of 34) and the time spent was 3.5 seconds.

The choice of the predicates declared as admissible also affects the amount of search. This influence can either be positive, reducing the number of considered refinements, as well as negative, increasing that number. By including, for example, the predicate *append/3* in the admissible predicate declaration, we obtain the same result after 86 refinements and 3.6 seconds.

If, instead of the three positive examples, SKIL is given the first positive example and one sketch, as shown below, the same program is synthesized after 9 refinements and in 2/3 of the time.

```
rv([],[]).    % positive example
sketch( rv([1,2,3],[3,2,1])←
        $P1([1,2,3],1,[2,3]), rv([2,3],[3,2]), $P2([3,2],1,[3,2,1]) ).
```

4.10 Limitations

As shown above, the SKIL system was able to synthesize a recursive definition for *rv/2* from three well-chosen positive examples. Whatever the presentation order of these three examples, the final result of SKIL always included the two clauses *c(12)* and *c(13)*. Some sequences give rise to a third clause, which is redundant in respect to the other two. The synthesis CPU time measured also fluctuates from experiment to experiment. In any case, this set of positive examples seems sufficient to induce the two relevant clauses.

We will now try a slightly different set of positive examples.

rv([],[]).
rv([1,2,3],[3,2,1]).
rv([4,5],[5,4]).

In this case, the program synthesized by SKIL is

c(12):rv(A,A)←
null(A).

c(14):rv(A,B)←
dest(A,C,D),
dest(D,E,F),
addlast(F,E,D),
addlast(D,C,B).

This program does not cover the example *rv([1,2,3],[3,2,1])* given. The search for a clause that covers this example terminates after exhausting the set of sketch refinements within the language bias. In particular, SKIL is not able to induce the recursive clause *c(13)* generated in the earlier run.

The recursive clause does not appear because the example *rv([2,3],[3,2])* is missing. In fact, SKIL has problems in generating recursive definitions from a set of positive examples which are not well-chosen, due to the strategy of searching for relational links. For this reason, we propose an iterative induction strategy that is capable of synthesizing

recursive clauses from sets of positive examples analogous to the one presented above. This is described in the next Chapter.

4.11 Related work

4.11.1 Linked terms

In 1977 Steven Vere [122] studied the problem of the induction of relational productions from examples in presence of a set of relevant facts (background knowledge) by linking terms in different literals. According to Vere, a *relational production* has the form $\alpha \leftarrow \beta$, where α and β are conjunctions of literals. In order to incorporate background knowledge literals into a conjunction of foreground literals, Vere proposed the notion of *association chain*. Two literals L_1, L_2 have an *association* if the $A_{i,j}(L_1,L_2)$ i-th term of L_1 is equal to j-th term of L_2. An *association chain* is a sequence of associations $A_{i_1,i_2}(L_1,L_2)$, $A_{i_3,i_4}(L_2,L_3),\ldots, A_{i_n,i_{n+1}}(L_{n-1},L_n)$, where for even r, $i_r \neq i_{r+1}$.

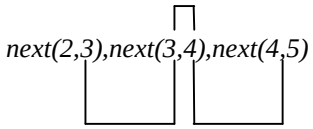

Figure 4.8:A Vere chain of associations example.

Figure 4.8 shows an example of one Vere's association chain. For the sake of clarity, we use Prolog notation instead of Vere's. A counterexample of an association chain is *next(2,3),next(3,4),odd(3)*. For a recent ILP approach to productions see [23].

Although association chains and the relational links described here have similar spirit, they represent different concepts. A relational link is defined in terms of input/output arguments and intends to connect two sets of terms: the set of input arguments and the set of output arguments. An association chain connects two literals. In an association

chain there is at most one connection between any two literals. Relational links are more complex since a literal may be connected to many others.

Richards and Mooney use *relational pathfinding* in the system FORTE [100] within a clause specialization method. The idea of this technique is to consider the set of terms in a logic program's Herbrand base as a hypergraph of terms linked by the relations (predicates) defined in the program. For example, given a positive example *uncle(arthur,charlotte)*, the search for a clause is made by expanding every term from the example. For that one considers the known data about the *parent/2* relation:

parent(cristopher,arthur).
parent(penelope,arthur).
parent(cristopher,victoria).
parent(penelope, victoria).
parent(victoria, charlotte).
parent(james, charlotte).
parent(victoria, colin).
parent(james, colin).

The expansion of the term *arthur* leads to the new terms {*christopher*, *penelope*} (facts *parent(cristopher,arthur)* and *parent(penelope,arthur)*). The expansion of the term *charlotte* leads to {*victoria*, *james*} (facts *parent(victoria, charlotte)* and *parent(james, charlotte)*). There is no intersection between the two term sets obtained in the expansion. By expanding the terms that resulted from the expansion of *arthur* we will obtain (finally) the term *victoria* (either fact *parent(cristopher,victoria)* or *parent(penelope, victoria)*). We have, therefore, an intersection between the set of terms obtained from *arthur*, {*chistopher*, *penelope*} $\cup$ {*victoria*}, and from *charlotte*, {*victoria*, *james*}, which corresponds to a relational path. This path can be arranged into

uncle(arthur,charlotte)$\leftarrow$*parent(christopher,arthur),*
parent(cristopher,victoria), parent(victoria, charlotte).

which corresponds to the clause

uncle(X,Y)←parent(Z,X),parent(Z,W),parent(W,Y).

The relational pathfinding (RP) technique is different from the relational linking technique used within the SKIL system in various aspects. In first place, SKIL strongly explores the input/output modes of the predicates involved in the definition. We can say that SKIL carries out a sort of directed relational pathfinding search. Secondly, in the FORTE system, when the RP method produces a clause which is over-general, the specialization of that clause is generated using a hill-climbing strategy, identical to FOIL [96]. In SKIL, the construction of a clause is made using only one specialization operator (Algorithm 3) which searches for relational links taking into account negative examples and integrity constraints. We have, therefore, a simpler clause construction algorithm which avoids the disadvantages of the hill-climbing method (cf. Section 3.4.5).

4.11.2 Generic programming knowledge

As already mentioned in Chapter 3, various generic programming knowledge representation formalisms have been proposed for the inductive construction of logic programs. Namely the *dependency graphs* by Wirth and O'Rorke [123], the *rule models* by Kietz and Wrobel [56], and the *clause schemata* by Feng and Muggleton [34]. Cohen [18] and Klingspor [58] also used the DCG notation to represent language bias in their systems.

The clause structure grammars used in SKIL are less expressive in comparison to other formalisms, particularly Cohen's DCGs, because they do not enable restricting the arguments of the literals of the induced clauses. The simplicity of CSGs is, however, advantageous particularly in what concerns construction and maintenance by the user.

4.12 Summary

The system SKIL synthesizes definite logic programs without functors or constants from a given specification, background knowledge and programming knowledge. The

specification contains positive examples, negative examples, integrity constraints, input/output mode and type declarations for the predicate to synthesize. Programming knowledge consists of clause structure grammars and algorithm sketches.

The synthesis of a logic program in SKIL proceeds by constructing one clause at a time. Each clause is constructed starting from an algorithm sketch associated to a given positive example. The construction strategy consolidates the sketch by seeking a relational link between the arguments of the literal in the head of the sketch. A candidate clause is extracted from the consolidated sketch through a variabilization operation. Candidates that cover any negative example are discarded. To find the appropriate sketch one explores the space of sketch refinements which is expanded using a sketch refinement operator. The clause structure grammar allows the definition of the structure of the clause to synthesize. The refinement operator takes this information into account.

The notion of sketch consolidation is formally defined and is related with the notion of sketch refinement. It is shown that the sketch refinement operator is complete with respect to operational consolidations of one sketch, assuming that no pruning is being done in the relevant sub-model. Assuming that the complete variabiliation is enforced we characterize the set of clauses that can be found by SKIL.

The main limitation of SKIL, which is shared by many other ILP systems, is the fact that it requires well-chosen examples in order to synthesize a recursive definition. This problem is addressed in the next Chapter where we introduce iterative induction.

5. Iterative Induction

This Chapter describes the problem of inducing recursive clauses and various approaches to this problem. We present the iterative induction method and the implemented system SKILit. This system is able to synthesize recursive definitions from sparse sets of positive examples. This solves the main limitation of system SKIL, presented in the previous Chapter.

5.1 Introduction

The induction of recursive definitions from positive examples is a difficult task for a typical ILP system. On the one hand, we have systems which require that the examples supplied are chosen with care (the so called good examples [63]). On the other hand, there are systems which do not require carefully chosen examples but only synthesize a small class of logic programs, which allows the use of specific strategies to search for recursive definitions ([1, 12, 49]. The SKILit system, presented in this Chapter, is

capable of synthesizing recursive definitions from examples which pose difficulties to other systems.

The SKILit system is an extension of the SKIL system, and uses an iterative induction strategy to synthesize recursive definitions from a set of examples chosen without prior knowledge of the required results.

5.2 Induction of recursive clauses

The possibility of defining concepts recursively in a concise and elegant way is one of the most attractive features of logic programming. Nevertheless, recursion is also a source of many practical and theoretical problems. In inductive logic programming, the problem of inducing recursive definitions from a set of naturally chosen examples is well known. In this Chapter we analyse this problem in detail, and describe our contribution to tackle it, by means of *iterative induction*.

The existing systems which induce recursive clauses from examples in a non-interactive fashion (without an oracle) can be divided in two groups according to approach they adopt. The first group includes approaches in which the positive examples do not affect the clause search space, which is explored exhaustively. Examples are used instead to define the stopping criterion (WiM [95], FORCE2 [12]). These can be regarded as brute force methods and are sometimes called model-driven methods. This approach has the advantage of being more robust with respect to variations in the initial set of examples, but the disadvantage of not exploiting those variations to accelerate the search.

The second group includes systems which generate the required clauses from positive examples and, in some cases, from background knowledge (SKIL and [1, 80, 82, 96]). In these systems, the examples are used to make heuristic-based decisions, thus reducing the initial search space. Therefore, these systems are less robust with respect to variations in the set of positive examples comparatively to the brute force methods. The

main advantage of this second approach is efficiency. These methods are sometimes called data-driven methods [1], as opposed to the model-driven ones.

For all the data-driven methods, it is important to consider a model M of the set of examples E^+ and of the background knowledge BK (that is, the set of facts that can be inferred from $E^+ \cup BK$).

5.2.1 Complete/sparse sets of examples

The FOIL system [12] can synthesize the definition of *member/2* if it is given all the facts about this predicate involving some list (e.g. *[1,2,3]*) and all its sub-structures. All these examples make possible the task of selecting the most appropriate literals. The results of FOIL get worse when the set of examples is not complete [97]. The reason why FOIL requires all these examples is that its heuristic function for selecting the best literal to add in each refinement step is computed in terms of the number of covered examples. Since coverage tests are extensional, the example sets must be complete. The GOLEM system has the same limitation.

Example 5.1: Clause *member(A,[B|Y])←member(A,Y)* only covers extensionally the positive example *member(2,[1,2,3])* if the positive example *member(2,[2,3])* is also given. ♦

Informally, and following Quinlan's terminology, we say that a *set of positive examples* is *complete* with respect to a set of clauses, if each example is extensionally covered by one of the clauses. A set of examples which is not complete is a *sparse set of examples*.

Example 5.2: Given the program that defines the predicate *member/2*,

```
member(A,[A|B]).                    (C1)
member(A,[B|C])←                    (C2)
    member(A,C).
```

A set of examples which includes *member(2,[1,3,2])* should also have *member(2,[3,2])* and *member(2,[2])* to be complete ♦

The fact that FOIL and GOLEM need a set of complete examples to synthesize the required set of clauses makes the task of inducing recursive programs hard. It is not expected, in a realistic situation, that the user supplies unnecessarily large sets of examples. ILP systems should be able to handle sparse sets of examples.

5.2.2 Basic representative set (BRS)

C. Ling [63] used the notion of *basic representative set* (BRS) to define what is a set of *good examples* for the induction of a logic program. For some ILP systems, a BRS is a necessary condition to synthesize a program. This is the case of all systems which employ extensional coverage tests. A program *P* can never extensionally cover a set of examples which does not include a BRS of *P*. This limitation includes systems such as FOIL, GOLEM, Progol, amongst others. The SKILit system does not require a basic representative set to synthesize a program.

A set of positive examples which is complete relatively to a set of clauses, contains at least one basic representative set (BRS) of those clauses.

Definition 5.1: A basic representative set of a program *P* is any set *S* of ground atoms obtained from a true ground instance (in the minimal model of *P*) of each clause $C \in P$. ♦

Since a clause in a logic program may have many true instances, the program may also have many basic representative sets.

Example 5.3: Given the program which defines the predicate *member/2*, (see Example 5.2) a basic representative set of that program is

{*member(1,[1,2]), member(4,[2,3,4]), member(4,[3,4])*}

which corresponds to the true instantiation

member(1,[1,2]).
member(4,[2,3,4])←member(4,[3,4]).

Another BRS for the same program is

{*member(3,[2,3,4]), member(3,[3,4])*}

The latter set has two examples only since *member(3,[3,4])* belongs to both clauses in the following instantiation.

member(3,[3,4]).
member(3,[2,3,4])←member(3,[3,4]).

If one of the examples is removed from any of the BRS above it is no longer a BRS.♦

Definition 5.2: Let *C* be a clause of a program *P*, a basic representative set of clause *C* with respect to *P*, denoted by *BRSC(C,P)*, is a set of ground atoms obtained from a ground instance of *C* which is true in the minimal model of *P*. The example which corresponds to the full instantiation of the head of *C* is a representative example of *C* with respect to *P*. ♦

By definition, a program's BRS may include examples of different predicates. However, for convenience, whenever we refer to a BRS of a program *P* defining predicate *p/k*, we will consider only the examples in the BRS which are relative to *p/k*. The elements of the BRS relative to other predicates are assumed to be extensionally or intensionally given as background knowledge.

5.2.3 Resolution path

Inductive synthesis may also take advantage when the given positive examples include at least a set of examples involved in one derivation. The set of atoms involved in the derivation of a fact is called *resolution path* or *resolution chain*.

The elements of a basic representative set relative to the same clause, belong to the same path (or chain) of resolution.

Definition 5.3: Let e be an example, P a program, and $D = ((R_1,C_1,\theta_1), (R_2,C_2,\theta_2), \ldots (R_n,C_n,\theta_n), \square)$, where $R_1 = \leftarrow e$ and $C_i \in P$, the derivation of e from P, the *resolution path* of e with respect to P, $RP(e,P)$ is the set of atoms

$$RP(e,P) = \bigcup_{i=1}^{n} atoms(R_i)\theta_1\theta_2\ldots\theta_n$$

where $atoms(R)$ represents the set of atoms in resolvent R. ♦

The resolution path of an example e with respect to a program P corresponds to the set of facts used to prove e from P. The elements of one basic representative set of a clause C are in the same resolution path. If e is a representative example of clause $C \in P$, and $D = ((\leftarrow e,C,\theta_1), \ldots , (R_n,C_n,\theta_n), \square)$, is a derivation of e from P, the set of literals in $C\theta_1\theta_2\ldots\theta_n$ is a $BRSC(C,P)$.

Example 5.4: Let us consider the program for *member/2* defined in Example 5.2 and the example *member(4,[3,2,4])*. To prove this fact we construct the derivation below.

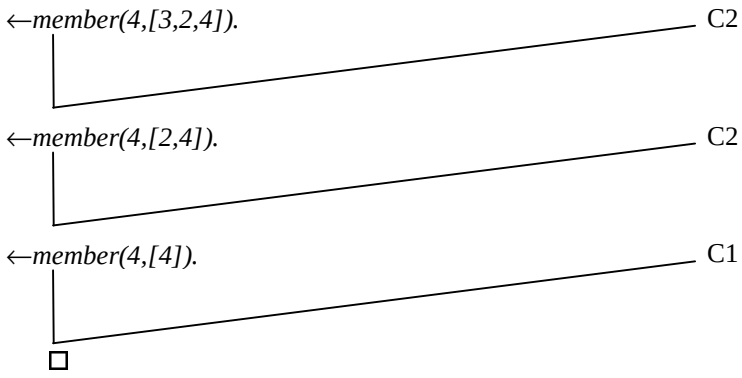

Figure 5.1: Derivation of a positive example.

This derivation is symbolically represented, omitting substitutions, by

$D = ((\leftarrow member(4,[3,2,4]),C2), (\leftarrow member(4,[2,4]),C2), (\leftarrow member(4,[4]),C1))$

The resolution path is now obtained by collecting the atoms in the resolvents of the derivation.

$$RP(member(4,[3,2,4]),P) = \{member(4,[3,2,4])\} \cup \{member(4,[2,4])\} \cup \{member(4,[4])\}$$

So, the examples in the resolution path of *member(4,[3,2,4])* are {*member(4,[3,2,4]), member(4,[2,4]), member(4,[4])*}.♦

Some methods, such as the *inversion of implication* by Muggleton [77] or the one used by system LOPSTER [60] (which employs a technique called *sub-unification*) do not require a BRS to induce a recursive clause. All they need is a representative example of that clause and another example in the resolution path representing the recursive literal. In the case of these two methods we are assuming that only one recursive literal is needed since the LOPSTER system only synthesizes clauses with at most one recursive literal. In the description of the algorithm for the inversion of implication this limitation is not mentioned, but it seems implicit.

Example 5.5: To induce the program in Example 5.2 it would be sufficient for a program like LOPSTER to have the examples *member(4,[3,2,4])* and *member(4,[4])*. We should stress that these two examples are not a BRS of the program.♦

For the system CRUSTACEAN (a follow-up of LOPSTER) the representative examples of a recursive clause do not have to belong to the same resolution path [1]. The technique used by this system to discover recursion consists in the analysis of the structure of the terms which are arguments of the examples.

Example 5.6: To induce the program in Example 5.2 the examples *member(4,[3,2,4])* and *member(1,[2,1])* would be sufficient for a program like CRUSTACEAN. Notice that the second example is not in the resolution path of the first one. ♦

5.3 Iterative induction

How does the SKIL system induce a recursive clause? Let us take a look at a particular situation. If it is given the positive examples

member(2,[3,2]).
member(2,[2]).

SKIL induces the program

member(A,B)←dest(B,C,D),member(A,D).
member(A,B)←dest(B,A,C).

This is a good result, since we have a recursive program that, given the definition of *dest/3* (Appendix A), covers the two positive examples. The second example is representative of the base clause, and the two are representative of the recursive clause. These two examples are a BRS of the induced program, and as a consequence, they are in the same resolution path.

If, however, we substitute one of the examples shown earlier by a somewhat different one, obtaining

member(2,[3,2]).
member(7,[7,1]).

the induced program is now

member(A,B)←dest(B,C,D),dest(D,A,E). (*Prop1*)
member(A,B)←dest(B,A,C). (*Prop2*)

We lost the recursive clause. However, this program induced by SKIL is not totally uninteresting. Even though the program is not recursive, each one of its clauses is a *property* of the concept of *member*. The first property, for example, says that every second element of a list is a member of that list. These properties which generalize the examples initially supplied to the system can now be exploited in the search for recursive clauses. Let us see how.

The reason why SKIL does not find the recursive clause from the examples {*member(2,[3,2])*, *member(7,[7,1])*} is as follows: To generate a recursive clause from the example *member(2,[3,2])* SKIL must construct the sketch

member(2,[3,2])←*dest([3,2],3,[2]),member(2,[2]).*

For that, it is necessary that each atom in the body of the sketch is in the model of {*member(7,[7,1])*}∪*BK*. However, this is not the case. The atom *member(2,[2])* is not in the model. For this reason, the recursive clause is not constructed.

The only reason for that atom not to be in the model, is that it is not one of the initial positive examples given to the system. Nevertheless, after the first passage of SKIL through the positive examples, the two properties *Prop1*, *Prop2* emerge. One of them covers the missing example (*member(2,[2])* ∈ *M*(*BK*∪{*Prop1*, *Prop2*})). In other words, the crucial example that was not in the initial data can be abduced by the SKIL system itself. As a consequence, SKIL now has the information to generate the recursive clause. Indeed, the recursive clause is generated during the second pass through the examples thanks to the properties generated earlier.

By generalizing this process we obtain an iterative algorithm which invokes SKIL in every iteration. We call this method *iterative induction*.

5.4 The SKILit algorithm

The SKILit algorithm (iterative SKIL) constructs logic programs using the iterative induction method. SKIL is invoked by SKILit as a sub-module which goes through the positive examples attempting to construct new clauses. Algorithm 5 describes this procedure in detail.

The SKILit algorithm starts with program P_0, which is initially empty. In the first iteration, SKILit constructs program P_1. The clauses in P_1 generalize some positive examples and are typically non-recursive. In general, it is difficult to introduce recursion

at this level, due to the lack of crucial positive examples among the given ones. It is likely, therefore, that the clauses in P_1 are defined using auxiliary predicates only (i.e., without recursive literals).

Procedure SKILit
input: E^+, E^- (positive and negative examples)
    BK (background knowledge).
output: P (logic program)
$i := 0$
$P_0 := \varnothing$
repeat
    $P_{i+1} := SKIL(E^+, E^-, P_i, BK)$
    $i := i+1$
until P_{i+1} does not contain new clauses with respect to P_i
$P := TC(P_{i+1}, BK, E^+, E^-)$
return P

Algorithm 5: Iterative induction

In a second iteration, program P_2 is induced. Here, it is more likely that recursion appears, since P_1 covers some crucial examples that were missing in the first iteration. Analogously, as P_2 covers more facts, other interesting recursive clauses may appear in the next iterations. The process stops when one of the iterations does not introduce new clauses. After the last iteration, the algorithm TC (theory compressor) is invoked. This module eliminates redundant clauses, which are typically properties induced in initial iterations and subsequently made redundant by recursive clauses.

5.4.1 Good examples

The method of iterative induction synthesizes program P by constructing a sequence of programs $P_0, P_1, \ldots, P_n$ where $P_0 = \varnothing$ and $P_n = P$. Each P_i is obtained by appending to P_{i-1} one or more clauses (with the exception of P_n which is equal to P_{n-1}). Therefore, and since we are dealing with definite programs, we have that

$$M(P_i \cup BK) \supseteq M(P_{i-1} \cup BK), \quad 1 \leq i \leq n$$

Since the model of $P_i \cup BK$ grows with i and, in each iteration i clause construction depends on the model of $P_{i-1} \cup BK \cup E^+$, the probability of synthesising the required recursive clause in a given iteration is at least as high as in the preceding iterations. But which initial set of examples should be given so that our method of iterative induction would induce the required recursive clause? How do we characterize a set of good examples?

As we saw on Section 5.3, iterative induction does not need a basic representative set of examples to synthesize a recursive clause. However, to synthesize a clause, the method needs all the atoms in a BRSC of that clause. Note that this does not imply that the set of initial examples must contain a BRSC. Let us then see which examples should be given.

Let us analyze the case of a recursive clause $C = (\ l_1 \leftarrow \ldots, l_2, \ldots)$ with a single recursive literal l_2. Let $\{e_1, e_2\}$ be the sub-set of a BRSC relative to predicate *p/k* defined in *C*. To synthesize *C*, iterative induction needs example e_1 and another example $\hat{e}_2$, which acts as substitute for e_2. Example $\hat{e}_2$ should be representative of a clause C_p that (together with *BK*) covers e_2 (the letter *p* was chosen since C_p is regarded as a property, and we will assume for now that C_p is non-recursive). Therefore, a set of good examples to synthesize *C* is $\{e_1, \hat{e}_2\}$. Iterative induction synthesizes C_p from $\hat{e}_2$ in iteration *i*. In iteration $i+1$ it synthesizes *C* from e_1 and C_p.

Example 5.7: Let us consider the following program *P*

member(A,B)←dest(B,A,C). (C1)
member(A,B)←dest(B,C,D),member(A,D). (C2)
dest([A|B],A,B). (C3)

One possible BRSC of C2 is { e_1 = *member(3,[1,2,3,4])*, e_2 = *member(3,[2,3,4])*}. A non-recursive clause C_p covering e_2 is

member(A,B)←dest(B,C,D),dest(D,A,E).

There are many examples covered by C_p which can figure as $\hat{e}_2$. One of them is, for instance, *member(5,[2,5])*. We then have a set of good examples for iterative induction { e_1 = *member(3,[1,2,3,4])*, $\hat{e}_2$ = *member(3,[2,5])*}. ♦

For each example e_2 there are several non-recursive clauses which cover that example. The example itself may be regarded as a ground unit clause. In order to characterize the acceptable examples $\hat{e}_2$, given a representative example e_2 of a clause we will show how to construct the non-recursive clause C_p.

Let P be a program and e_2 an example covered by that program. We can, by applying resolution to the clauses of P, obtain a non-recursive clause C_p which covers e_2. Let D be a refutation $((\leftarrow e_2, C_1, \theta_1), (R_2, C_2, \theta_2), \ldots, (R_n, C_n, \theta_n), \square)$ of e_2 from P. The clause C_p is obtained by transforming clause C_1 according to the sequence of derivation steps in D, skipping those which resolve recursive literals. The process is described in detail below.

First we remove from D all the derivation steps involving clauses which are not defining predicate *p/k*. We also remove the first derivation step from D. In what is now the first derivation step (R_j, C_j, θ_j), we replace R_j with C_1. We resolve a negative literal of C_1 with the positive literal of C_j thus obtaining the derivation step (C_1, C_j, σ_j). By applying the remaining steps of D we get as result a clause C_p. This is a non-recursive clause covering e_2.

Example 5.8: Continuing with Example 5.7 we will show how a clause C_p is constructed from P. The derivation D of e_2, omitting the substitutions, is

((←*member(3,[2,3,4])*, C2),
(←*dest([2,3,4],A,B),member(3,B)*, C3),
(←*member(3,[3,4])*, C1), □)
(see Figure 5.2).

We now remove the first step involving clause (C2) to the derivation. We do the same to the step involving C3 since this clause is not defining *member/2*. We are left with the step involving C1. By resolving C2 with C1 we obtain the non-recursive clause C_p covering e_2.

member(A,B)←dest(B,C,D),dest(D,A,E).

♦

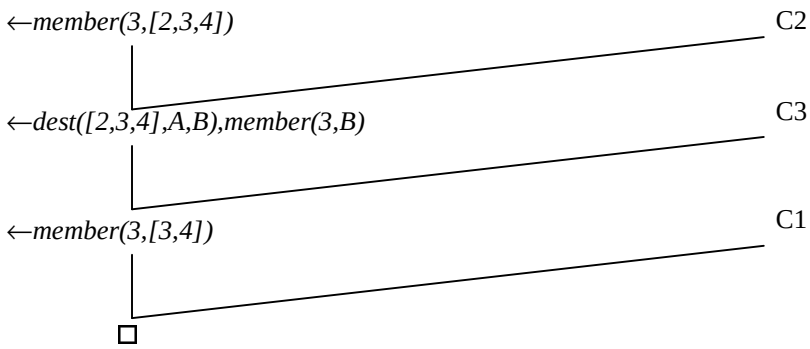

Figure 5.2: Derivation *D* of the example e_2.

An important question must be answered now:

- Given any example $\hat{e}_2$ covered by clause C_p as constructed above does iterative induction always give a clause covering e_2?

In general, given an example $\hat{e}_2$ covered by a clause covering another example e_2, the method of iterative induction may synthesize a clause $\hat{C}_p$ not covering e_2 (although experience tells us that in most cases it does). This is because the algorithm that constructs the programs in each iteration (SKIL, Algorithm 1) uses a *covering strategy*. If an example is covered SKIL does not try to find another alternative clause to cover it. Therefore the first clause found is the one that stays. This problem caused by the covering strategy suggests a more powerful (yet heavier) non-covering strategy. This alternative strategy will be described below in Section 5.4.2.

The analysis done so far applies to a clause with a single recursive literal. If clause *C* has more than one recursive literal we need an example analogous to $\hat{e}_2$ for each of those literals. Since the BRSC of a clause with *k* recursive literals $C = (l_1 \leftarrow \ldots, l_2, \ldots, l_{k+1}, \ldots)$ contains $k+1$ examples $\{e_1, e_2, \ldots, e_{k+1}\}$, iterative induction needs a set of examples $\{e_1, \hat{e}_2, \ldots, \hat{e}_{k+1}\}$ to be successful. Each example $\hat{e}_i$ represents a clause C_i covering e_i , $2 \leq i \leq k+1$.

For each BRSC $\{e_1, e_2, \ldots, e_{k+1}\}$ of a clause *C* in a program *P* we have a family of sets of good examples. We call each one of these sets a BRSCI (clause basic representative set of examples for iterative induction). Each BRSCI $\{e_1, \hat{e}_2, \ldots, \hat{e}_{k+1}\}$ is obtained from the BRSC by replacing one or more examples e_i $2 \leq i \leq k+1$, with an example $\hat{e}_i$ covered by a non-recursive clause obtained by resolution from *P* as described above.

Note however, that since SKILit is iterative, the auxiliary property C_p may itself be a recursive clause. In that case the set of good examples to generate the target recursive clause *C* must include one set of good examples to generate C_p.

Example 5.9: We can synthesize a recursive definition of *member/2* from the following examples:

member(2,[1,3,2,4]).
member(5,[5,6]).
member(6,[1,2,3,4,5,6]).

-member(2,[]).
-member(2,[3]).
-member(2,[1,4,3]).
-member(2,[1,4]).

Using the CSG ´decomp_test_rec_comp_2' in the first iteration it is only possible to generate the non-recursive clause

member(A,[A|B]).

Its representative example is *member(5,[5,6])*. In the second iteration SKILit obtains the clause

member(A,[B,C|D])←member(A,D).

This is a recursive property of *member/2* generated from example *member(2,[1,3,2,4])* and the first clause. The two clauses still do not cover example *member(6,[1,2,3,4,5,6])*. From this example and from the recursive property another recursive clause appears in the third iteration:

member(A,[B|C])←member(A,C).

The three initial positive examples are a set of good examples to synthesize this clause.♦

Since program *P* is not known before being synthesized, how can we construct a BRSCI? A good strategy is to give a series of positive examples whose input terms increase in complexity (in case we are in presence of structured terms, such as lists) or in value (in case we are dealing with an ordered domain, such as integers) starting with the most simple case (list *[]*, integer *0*) and ending up with reasonably complex terms (lists of length 4 or less, integers up to *4*). For each level of complexity we should provide examples which represent different cases. For example *sort([1,2],[1,2])* and *sort([2,1],[1,2])* represent two possible cases for sorting lists of length 2. One exchanges the elements of the input list and the other does not.

5.4.2 Pure iterative strategy

As we saw above, when the default covering strategy is used we cannot guarantee that SKILit always finds some clause C_p given any example $\hat{e}_2$ covered by that clause. For that reason we introduce here a new iterative strategy.

At each iteration, SKILit tries to construct a new clause for each positive example, covered or uncovered. Note that with the covering strategy SKILit does not use covered examples to generate new clauses. The process stops when no new clauses are found in one iteration. Termination is guaranteed if the clause language is finite, as it usually is. In any case it can be made finite by defining an appropriate clause structure grammar.

We call this procedure the *pure iterative strategy*. If the complete variabilization method is in use each example may give in each iteration a set of clauses instead of just one. The induction strategy is chosen through a declaration in the specification and it corresponds to turning on or off the covering condition in Algorithm 1 (clause constructor).

Example 5.10: Here we show how the covering and pure iterative strategies may have different results. The task consists of the multiple synthesis of predicates *sort/2* and *insert/3*. The specification contains information relative to both predicates (see Section 5.5.3). We give the same input to SKILit with each of the strategies on and compare the results.

Input:

```
sort([3,2,1],[1,2,3]).          environment( list ).
insert(2,[1],[1,2]).            csg( decomp_test_rec1_comp_2 ).
insert(6,[],[6]).               adm_predicates( sort/2,
sort([],[]).                        [dest/3,const/3,insert/3,sort/2,'<'/2,null/1]).
insert(1,[2],[1,2]).            adm_predicates( insert/3,
sort([5,4],[4,5]).                  [dest/3,const/3,'<'/2,null/1,insert/3]).

-insert(2,[1],[2,1]).           % choose strategy as appropriate.
-insert(1,[2],[2,1]).           strategy( pure_iterative).
-insert(3,[1,2],[3,1,2]).       strategy(covering).
-insert(3,[1,2],[1,3,2]).
-sort([1,2],[2,1]).
-sort([1,3,2],[1,3,2]).
-sort([3,2,1],[2,3,1]).
-sort([3,2,4,1],[2,3,4,1]).
-sort([2,3,1],[2,3,1]).
```

Output, covering strategy:

```
sort([],[]).
sort([A,B|C],D)←insert(B,[A,B|C],E), insert(A,[B|C],D).

insert(A,[B],[B,A]) ←B<A.
insert(A,[],[A]).
insert(A,[B|C],[A,B|C]) ←A<B.
```

Number of iterations: 2
575 refinements (total)
4.64 secs

Output, pure iterative strategy

sort([],[]).
sort([A,B|C],D) ←insert(B,[A,B|C],E), insert(A,[B|C],D).
sort([A,B],C) ←insert(A,[B],C).
sort([A|B],C) ←sort(B,D), insert(A,D,C).

insert(A,[B],[B,A]) ← B<A.
insert(A,[],[A]).
insert(A,[B|C],[A,B|C]) ←A<B.
insert(A,[B|C],[B|D]) ←B<A, insert(A,C,D).
insert(A,[],[A]).
insert(A,[B],[A,B]) ← A<B.
insert(A,[B],[B|C]) ← B<A, insert(A,[],C).

Number of iterations: 5
2230 refinements (total)
35.37 secs

These results are produced without TC. Note that, given the positive examples in that specific order, the covering strategy is able to find a non-recursive program that covers all of them except *sort([3,2,1],[1,2,3])* (there is no clause in the search space that covers this example). The pure iterative strategy finds alternative clauses, and among them some recursive ones. The final program (in bold) can be found by a compression module like TC. Note that heuristic pruning of the relevant sub-model and the simple variabilization technique were used in this example. ♦

We now characterize the set of clauses that SKILit can generate using pure iterative induction.

Theorem 5.1: Let S be a sketch, P a program, G a clause structure grammar and C a clause. If $C\theta \angle S$ ($C\theta$ is a consolidation of S) for some substitution θ, and the set of

clauses accepted by *G* is finite, then SKILit, with the pure iterative strategy, no pruning heuristics and complete variabilization outputs *C* in finite time given *S*, *P* and *G*.

Proof: If $C\theta \angle S$ then, by the completeness of the sketch refinement operator, $C\theta \in \rho^*(S)$. Since *G* accepts only a finite set of clauses then $\rho^*(S)$ is finite. Using the pure iterative strategy SKILit constructs all the consolidations of *S* and eventually finds the sketch $C\theta$. One of its variabilizations is necessarily *C*. Therefore *C* is output in finite time by SKILit with the pure iterative strategy.♦

As a consequence, given the set of examples $\{e_1, \hat{e}_2\}$, and background knowledge *BK* if there is a non-recursive clause C_p that together with *BK* covers $\hat{e}_2$ then SKILit generates that clause. If there is a recursive clause *C* with a BRSC $\{e_1, e_2\}$ and $C_p \cup BK$ covers e_2 then SKILit generates *C* from $\{e_1, \hat{e}_2\}$. Similarly, we can have multiple recursive literals in clause *C* or a chain of recursive properties as in Example 5.9.

5.4.3 SKILit architecture

The diagram in Figure 5.3 shows the relationship between the main modules in the SKILit system. Each arrow shows module dependencies. The module at the top is SKILit (Algorithm 5), which iteratively calls the sub-system SKIL (Algorithm 1) and also employs the theory compressor TC.

The TC module is the program compressor (or theory compressor). The logic program obtained through iterative induction may contain clauses (properties) which, although useful to the induction process, are not necessary in the final program. Some may even be undesirable, causing non-termination. In any case, it is important to eliminate these clauses for reasons of efficiency and readability. This is the role of TC.

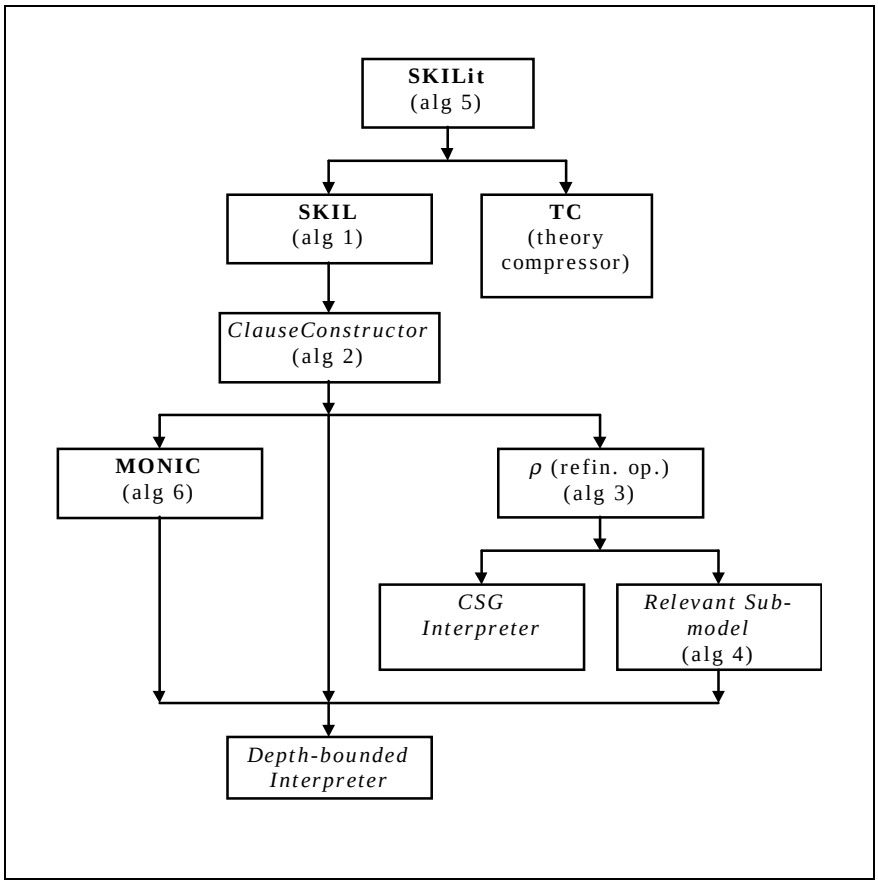

Figure 5.3: The SKILit system architecture

The Theory Compressor gets as input a program induced by the iterative induction method, and selects a subset of its clauses. For that, it uses a strategy of *sub-theory selection*. It identifies useful sub-programs of the whole induced program. From the combination of those sub-programs, it constructs a final program which maximises a combination of criteria. The criteria are defined in terms of positive example coverage, compression, and solution length [15]. The theory compressor is not described in detail in this thesis.

5.5 Example sessions

Here we show examples that demonstrate SKILit's abilities in the synthesis of recursive predicates from sparse sets of examples. The predicate *union/3* has two interdependent

recursive clauses. The predicate *qsort/2* has one clause with two recursive literals. We also show how to use SKILit in a multi-predicate synthesis task. For these sessions, system SKILit used the covering strategy, heuristic pruning and simple variabilization.

5.5.1 Synthesis of union/3

We show an example of synthesis of a definition of the predicate *union/3* from positive and negative examples and programming knowledge. The given examples have been chosen following the strategy described earlier, varying the complexity of input terms. Notice, however, that the positive examples are not a basic representative set of the program, neither belong to the same resolution path.

Specification:

mode(union(+,+,-)).
type(union(list,list,list)).

union([],[2,3],[2,3]).
union([2],[2,3],[2,3]).
union([2],[3,4],[2,3,4]).
union([2,3],[4,2,5],[3,4,2,5]).
union([2,3],[4,5],[2,3,4,5]).

-union([2],[3,4],[3,4]).
-union([2,3],[2],[2]).
-union([2],[1,2],[2,1,2]).
-union([2,3],[4],[2,4]).

Background and programming knowledge:

```
background_knowledge(list).
adm_predicates( union/3,
      [dest/3,const/3,null/1,union/3,member/2,notmember/2]).
clause_structure(decomp_test_rec_comp_2).
```

Synthesized program (before elimination of redundant clauses)**:**

```
c(17):union([],A,A).
c(18):union([A],B,B)←                          % redundant
   member(A,B).
c(19):union([A],B,[A|B])←                      % redundant
   notmember(A,B).
```

c(20):union([A|B],C,D)←
member(A,C),
union(B,C,D).
c(21):union([A|B],C,[A|D])←
notmember(A,C),
union(B,C,D).

Number of iterations: 2
408 refinements (total)
18.22 secs

The synthesized program contains 3 clauses (c(17), c(20), c(21)), two of which are recursive. The module of theory compression TC, eliminates two redundant clauses (*c(18)* and *c(19)*). These are intermediate clauses which serve as properties and are fundamental in the synthesis of the recursive clauses. The clauses are presented unflattened. Unflattening is done by SKILit only to improve the readability of the results.

Predicate *notmember/2* is employed since SKILit does not induce clauses with negated literals. Even though this limitation of SKILit would be simple to overcome, the search would become somewhat heavier and the results could be different.

5.5.2 Synthesis of qsort/2

The synthesis of the definition of the *quicksort* sorting algorithm is a classical challenge for systems which synthesize recursive definitions. For this reason, we show the result obtained by SKILit on this task. What determines that the sorting algorithm constructed by SKILit is *quicksort*, and not another one, are the admissible auxiliary predicates and the clause structure grammar.

Specification:

mode(qsort(+,-)).
type(qsort(list,list)).

qsort([],[]).
qsort([3,1],[1,3]).
qsort([3,2,5,1,4],[1,2,3,4,5]).

-qsort([2,1],[2,1]).
-qsort([1,2],[2,1]).
-qsort([3,1,2],[1,3,2]).
-qsort([3,2,1],[2,1,3,1]).
-qsort([2,3,1],[2,3,1]).
-qsort([3,2,1],[2,1,3]).

Background and programming knowledge:

```
background_knowledge( list ).
adm_predicates( qsort/2,
        [dest/3,const/3,partb/4,appendb/3,qsort/2,'<'/2,null/1]).
clause_structure(decomp_test_rec_comp_2). % Appendix C

%parameters
max_num_of_refinement_nodes(2500).
```

Synthesized program (before elimination of redundant clauses)**:**

```
c(18):qsort([],[]).
c(19):qsort([A,B],[B,A])←                                   % redundant
   B<A.
c(20):qsort([A|B],C)←
   partb(A,B,D,E),
   qsort(D,F),
   qsort(E,G),
   appendb(F,[A|G],C).

Number of iterations: 2
2780 refinements (total)
386.415 secs
```

Once more, an intermediate property is synthesized (clause c(19)), which is eliminated by the TC module.

5.5.3 Multi-predicate synthesis

Here we show how SKILit can synthesize a two-predicate program. The predicates to define are *sort/2* and *insert/3*. The specification includes positive and negative examples from both predicates as well as mode and type declarations. The clause structure grammar is one, but each predicate has a different list of auxiliary predicates. These are all defined in the background knowledge *list*.

Input:

mode(sort(+,-)).
type(sort(list,list)).

mode(insert(+,+,-)).
type(insert(int,list,list)).

sort([3,2,1],[1,2,3]).
insert(6,[],[6]).
sort([],[]).
insert(1,[2],[1,2]).
sort([5,4],[4,5]).
insert(2,[1],[1,2]).

-insert(2,[1],[2,1]).
-insert(1,[2],[2,1]).
-insert(3,[1,2],[3,1,2]).
-insert(3,[1,2],[1,3,2]).
-sort([1,2],[2,1]).
-sort([1,3,2],[1,3,2]).
-sort([3,2,1],[2,3,1]).
-sort([3,2,4,1],[2,3,4,

background_knowledge(list).
clause_structure(decomp_test_rec1_comp_2).
adm_predicates(sort/2,[dest/3,const/3,insert/3,sort/2,'<'/2,null/1]).
adm_predicates(insert/3,[dest/3,const/3,'<'/2,null/1,insert/3]).

In the output (SKILit's trace) we can see the order by which clauses have been generated. In the first iteration we have base clauses and useful properties for both predicates. The recursive clause of *insert/3* is also synthesized in the first iteration. In the second iteration we have the recursive clause for *sort/2*.

Output:

Iteration #1

example to cover: sort([3,2,1],[1,2,3])
empty queue.

example to cover: insert(6,[],[6])
clause c(26) generated after 7 refinements:
insert(A,[],[A]).

example to cover: sort([],[])
clause c(27) generated after 2 refinements:
sort([],[]).

example to cover: insert(1,[2],[1,2])

clause c(28) generated after 47 refinements:
insert(A,[B|C],[A,B|C])←
A<B.

example to cover: sort([5,4],[4,5])
clause c(29) generated after 101 refinements:
sort([A,B],C)←
insert(B,[A],C).

example to cover: insert(2,[1],[1,2])
clause c(30) generated after 105 refinements:
insert(A,[B|C],[B|D])←
B<A,
insert(A,C,D).

Iteration #2

example to cover: sort([3,2,1],[1,2,3])
clause c(31) generated after 14 refinements:
sort([A|B],C)←
sort(B,D),
insert(A,D,C).

example to cover: insert(6,[],[6])
example covered by existing clause c(26)

example to cover: sort([],[])
example covered by existing clause c(27)

example to cover: insert(1,[2],[1,2])
example covered by existing clause c(28)

example to cover: sort([5,4],[4,5])
example covered by existing clause c(29)

example to cover: insert(2,[1],[1,2])
example covered by existing clause c(30)

Synthesized Program:

c(27):sort([],[]).

```
c(29):sort([A,B],C)←                                    % redundant
   insert(B,[A],C).

c(31):sort([A|B],C) ←
   sort(B,D),
   insert(A,D,C).

c(26):insert(A,[],[A]).

c(28):insert(A,[B|C],[A,B|C]) ←
   A<B.

c(30):insert(A,[B|C],[B|D]) ←
   B<A,
   insert(A,C,D).

Number of iterations: 2
351 refinements (total)
```

Although SKILit is able to perform multiple predicate synthesis, we have not carefully evaluated our methodology in this sort of tasks. In particular, no systematic empirical evaluation was done to quantify the succes and limitations of our approach to multiple predicate synthesis. We intend to do this in the future.

5.6 Limitations

In this section we describe the main limitations of the inductive synthesis approach.

5.6.1 Specific programs

The programs synthesized by SKILit are sometimes more specific than those which would be constructed by other systems able to induce recursive definitions from sparse sets of positive examples, as it is the case of CRUSTACEAN.

Example 5.11: Given the positive examples

member(2,[1,2,3]).
member(3,[5,4,3]).

SKILit generates program:

member(X,[Y,X|Z]).
member(X,[Y|Z])←member(X,Z).

Other systems, such as CRUSTACEAN, synthesize a more general program.

member(X,[X|Z]).
member(X,[Y|Z])←member(X,Z).

SKILit's program does not cover the example *member(2,[2,1])*, while the second program does.♦

This feature of SKILit can be regarded as a limitation relatively to other systems. However, there is no guarantee that the program that the user has in mind is the more general one, instead of the other. In other words, this characteristic of SKILit is sometimes a limitation, but other times it can be an advantage. An evidence of that is that SKILit competes well with CRUSTACEAN, as we can see in Section 6.4.1.

5.6.2 Variable splitting

The breadth-first search performed by SKILit while attempting to construct a clause, is sustainable due to some options taken to reduce the search space. One of these options involves the transformation of constants into variables, which we call variabilization, occurring in Algorithm 2. Given a fully instantiated clause, the aim of variabilization is to find one or more clauses which have the initial one as an instance.

The process of simple variabilization currently implemented in SKILit is efficient (see Section 4.7.1.1). Moreover, the variabilization of a particular sketch results in only one clause, which avoids the problem usually referred to as *variable splitting* [102]. This fact helps controlling the branching factor of the search tree. The disadvantage of this simple variabilization process is that it does not take into account the fact that the same constant

may correspond to two different variables. This may prevent the synthesis of some desirable clauses. As it has been mentioned before, the complete variabilization process solves this problem.

Example 5.12: According to the process employed by SKILit, the variabilization of the clause

p(a,z)←q(a,c),t(a,c,z). (1)

is obtained by replacing all the occurrences of a constant with the same variable. Different variables correspond to different constants. The result is

p(A,Z)←q(A,C),t(A,C,Z).

This variabilization process is simple and efficient, and has only one clause as a result. However, the clause

p(A,Z)←q(B,C),t(A,C,Z).

also has the clause (1) as instance. A variabilization process giving, as a result, the set of all the clauses with clause (1) as an instance would produce a long list of clauses including

p(A,Z)←q(A,C),t(B,C,Z).
p(A,B)←q(C,D),t(E,F,G).

etc.♦

The simple variabilization process avoids the problem usually referred to as *variable splitting* [102]. Therefore, we can drastically reduce the branching factor of the refinement tree. As noted earlier, the complete variabilization process could be given as an option.

5.7 Related work

5.7.1 Closed-loop learning

Michalski describes in [68] the notion of a *closed-loop learning system* as a learning system able to use the learned concepts as input in another learning phase. If the learnt concepts are not internally exploited by the system, then it is called an *open-loop system*. Michalski stresses that contrary to human learning systems, the machine learning systems are typically open-loop systems. The iterative induction method follows the philosophy of closed-loop systems in the process of learning recursive clauses. The clauses learnt in initial iterations are employed by the learning process in the following ones.

One of the three induction strategies which Shapiro's MIS system [109] can use is the *adaptive strategy*. In this case, MIS works as a closed-loop system, using the induced clauses to aid the induction of new clauses, as it happens in iterative induction. In the MIS system, the search for clauses is exhaustive, whilst in SKILit the search is guided by the examples through the sketch consolidation strategy. On the other hand, SKILit can start the search for a clause starting from any sketch, whilst MIS always sets out from the empty clause.

The CHILLIN [125] and RTL systems [40] also use an iterative strategy for the induction of recursive clauses. However, there seem to exist significant differences between these approaches and the iterative induction method we propose.

The system CHILLIN interleaves a clause generalization phase with a clause specialization phase. These two phases are repeated until no further compaction of the program is possible. The generalization phase uses the least general generalization operator. The specialization phase employs a top-down search guided by a heuristic similar to that of FOIL [96]. Even though this heuristic works well with a relatively large set of examples, it does not seem adequate for the synthesis of recursive definitions from a small set of examples.

The RTL system uses an iterative method for the definition of recursive definitions. In the first step, the system produces non-recursive definitions, which are subsequently transformed into recursive ones. SKILit proceeds in an analogous way, since it frequently also starts by first producing non-recursive definitions. However, it is difficult to foresee which results RTL would obtain with small sets of positive examples. In [40] no experimental results are given which could answer this question. However, we believe that RTL would not give very good results with small example sets since it also employs a FOIL-like heuristic.

5.7.2 Sparse example sets

We have already referred here approaches to the synthesis of recursive clauses from a sparse small set of positive examples (Section 5.2). Systems like FORCE2 [12], LOPSTER [60], and CRUSTACEAN [1] are devoted to that specific problem. Although efficient, these systems induce a very restricted class of programs. CRUSTACEAN, for instance, induces programs of the form

$p(\ldots)$.
$p(\ldots) \leftarrow p(\ldots)$.

and does not allow the use of background knowledge predicates. The class of programs synthesizable by FORCE2 is described by

$p(\ldots) \leftarrow q_1(\ldots), \ldots q_n(\ldots)$.
$p(\ldots) \leftarrow r_1(\ldots), \ldots, r_m(\ldots), p(\ldots)$.

Each predicate q_i and r_j is a background knowledge predicate. An important negative aspect of FORCE2 is that the user must indicate which examples are covered by the base clause and which are not.

As it was described SKILit can induce programs with variable number of recursive and non-recursive clauses and a variable number of recursive literals. Another important feature of SKILit which is not shared by those approaches is that its result is not

necessarily a recursive program (unless the clause structure grammar imposes that). A non recursive solution is output whenever appropriate. Recursive solutions appear only if they involve shorter clauses than non-recursive ones.

Systems TIM [49] and SMART [74] also present approaches to the problem of learning recursive definitions from sparse example sets. They were not known, however, before SKILit was first presented [53].

The system SMART of Mofizur et al. is able to induce theories consisting of one base clause and one recursive clause. While the base clause is induced using a term decomposition process akin to CRUSTACEAN's, the recursive clause is built in a top down fashion following somewhat MIS [109]. The system restricts the search by examining variable dependencies. The system is capable of learning the definitions of various list processing predicates from small sets of examples. The class of target programs is, however, more restricted than in SKILit.

System TIM, instead of looking for regularities within the terms in the examples as CRUSTACEAN, constructs explanations of the examples in terms of background knowledge. These explanations are referred to as *saturations*. After saturations have been constructed for all positive examples TIM looks for regularities in pairs of saturations and uses those to generate the recursive clause. The search for common path structure is expensive, but it can lead to quite good results. Comparing experimental results of TIM and SKILit conducted in similar conditions (but not necessarily the same), we conclude that there is no clear winner. In any case some programs are outside TIM's scope altogether. TIM constructs definitions syntactically similar to the ones of FORCE2.

Notice that both SMART and TIM assume that the solution is a recursive program, contrary to SKILit.

5.8 Summary

System SKILit is an extension of the SKIL system, presented earlier. SKILit uses an iterative induction strategy which enables the synthesis of recursive definitions from a sparse set of positive examples.

The iterative induction consists in repeatedly invoking SKIL using the clauses produced in one iteration as input for the subsequent iterations. In the first iterations, non-recursive clauses that generalize some of the positive example typically arise. These clauses are called properties and serve to support the introduction of recursive literals in the following iterations.

The iterative induction strategy overcomes the problem of inducing recursive clauses from sparse sets of positive examples. We characterized the sets of good examples for the synthesis of recursive clauses using iterative induction and described two alternative strategies: the covering strategy and the pure iterative strategy. We showed which clauses are output by SKILit using the pure iterative strategy.

6. Empirical Evaluation

In this Chapter, we present an empirical evaluation of the SKILit system. We describe an evaluation methodology and show results of some experiments. Comparative experiments between SKILit and other systems are also presented.

This Chapter summarizes the experiments conducted to obtain an empirical evaluation of the SKILit system. The objective of the evaluation is to provide supporting evidence concerning the advantages and disadvantages of the SKILit methodology and system. More specifically, we want to validate the system adequacy to the synthesis of recursive logic programs from sparse sets of positive examples. The experimental methodology described here attempts to simulate a human user of a synthesis system who does not know the target programs beforehand.

The questions we want to answer are the following:

- What is the performance of SKILit in the synthesis of (recursive) definitions from sparse sets of positive examples?
- How does SKILit compare to other state-of-the-art ILP systems?

To answer these questions we used the experimental methodology described in the next Section. Until recently, proposed ILP systems were usually not systematically tested following an experimental methodology. Instead, the virtues and weaknesses of the systems were described with the help of some chosen examples [20,38,109]. This is obviously not sufficient. However, some recent approaches to the problem of recursive definition induction from sparse sets of examples adopted a systematic evaluation methodology in order to test the robustness of the proposed systems to variations in the choice of the positive and negative examples [1, 49, 125].

6.1 Experimental methodology

We have decided to adapt an evaluation strategy which is commonly used in ML to the needs of ILP. Each experiment consists of making the inductive synthesis system (for example, SKILit) run through a set of positive and negative examples, called the *training set*. The resulting logic program is then evaluated through another set of positive and negative examples, called the *test set*. The evaluation through the test set is essentially done in terms of the number of positive and negative examples covered by the induced program (Figure 6.1).

To evaluate the robustness of the synthesis system with respect to the choice of training examples, a series of experiments (10 or 20 repetitions) are performed. For each experiment, the training set is randomly constructed from a universe of positive examples and a universe of negative examples which are defined a priori. The training set is given as a specification to SKILit. The resulting synthesized program is then evaluated on a test set. Each test set is also randomly constructed from universes of positive and negative examples. While a new training set is constructed for each experiment, the test set remains the same for the whole series of experiments for the same predicate.

The *universe of positive examples* of a given relation is a subset of elements of that relation. The probability of extracting each example in that universe is also established.

The *universe of negative examples* contains elements which do not belong to the relation.

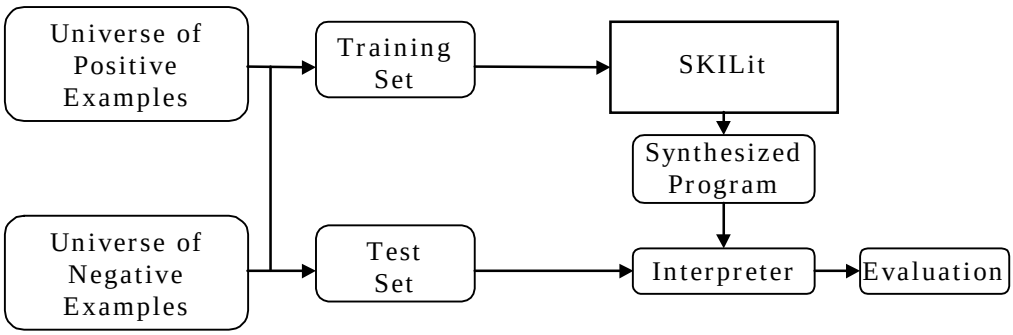

Figure 6.1: Experimental Methodology

The evaluation of the system's performance in one experiment consists of calculating the success rate of the induced program with respect to the test set, measuring the CPU[5] time spent by the system during the induction. For each series of experiments we also measure the percentage of programs which have success rate equal to 1 (i.e., are error free). This measurement is called the percentage of test-perfect programs.

Other measurements are made for each experiment such as counting the number of clauses in each program. However, such results are not reported here.

6.1.1 Success rate, test-perfect programs and CPU time

The *success rate* of a logic program P with respect to a test set TS with $\#E^-$ negative examples and $\#E^+$ positive examples is

$$sr(P,TS) = \frac{cov(P,E^+)+(\#E^- - cov(P,E^-))}{\#E^+ + \#E^-}$$

[5] Central Processor Unit.

where $cov(P,E^*)$ is the number of positive or negative examples covered by P. The coverage test is intensional, and is conducted with a depth-bounded interpreter (Section 4.7.4).

After measuring the success rate of each induced program P_1, P_2,…, P_n, using a test set TS in a series of n experiments, we can calculate the *percentage of test-perfect programs*.

$$tpp(SynthProgs,TS) = \frac{\#(\{P \mid sr(P,TS)=1,\ P \in SynthProgs\ \})}{n} \times 100\%$$

where $SynthProgs = \{ P_1, P_2,\ldots, P_n \}$, and # is the cardinality of a set.

The percentage of test-perfect programs estimates the probability of obtaining a correct program by running SKILit once. In our opinion it is important to consider the percentage of test-perfect programs, and not just the average success rate. We want to distinguish a situation where a system synthesizes an acceptable program in 90% of the cases from the one where a program with a success rate of 0.90 is always synthesized. In our opinion, the first situation is preferable when the application is program synthesis.

The CPU time spent for each experiment was measured on a SUN computer with a SPARC10 processor. SKILit was implemented with the Yap Prolog compiler [2].

6.1.2 The universe of positive examples

The distribution of positive examples $p(X_1,\ldots,X_k)$ of a relation p/k in a universe of positive examples is defined in terms of the distributions of the types of each argument X_i. A type which corresponds to a set of non-structured terms, such as *int* (0, 1, 2,…, 9), has a uniform distribution over a finite subset. A structured type, such as *list*, has a uniform distribution over its dimension up to a certain limit. In the case of a list, the dimension is its length. A list with a length greater then the limit is not considered. The length of list *[]* is 0 and the length of a list *[X|Y]* is 1+length of *Y*. The universe of

positive examples involving structures with dimension smaller or equal to 4 is called U4(+). The universe of positive examples involving structures with dimensions greater or equal to 3 and less or equal to 5, is called U3:5(+). To choose the subterms of a structured term we consider the type of the subterm.

We can also have positive examples where only the input arguments are restricted in dimension. The universe U2i(+) is made of positive examples where the maximum dimension of input arguments is 2.

The extraction of a positive example $p(X_1,\ldots,X_k)$ is made by extracting every term X_i of type T_i according to the distribution of T_i. The $p(X_1,\ldots,X_k)$ which do not belong to the relation are obviously not considered. The task of extracting a positive example is simplified by taking advantage of the predicate *p/k* mode declaration. This way, one only extracts the input terms, whereby the output terms are determined by these. In case the predicate is not deterministic, the output terms should be randomly chosen among the various possible answers.

6.1.3 The universe of negative examples

In our experiments, we initially considered two sorts of negative examples: random and 'near misses'. We will now describe the two sorts of examples.

A random negative example $p(X_1,\ldots,X_k)$ is generated in an analogous way to a positive example, i.e., extracting every term X_i according to the distribution of its type T_i and checking that it is in fact a negative example of the relation. The universe of random negative examples involving lists with lengths equal or less than 4 is called U4(-).

The negative examples we call 'near misses' are syntactically close to positive examples, but lying out of the relation. The extraction of a 'near miss' is made by syntactically corrupting a positive example and checking that the resulting fact does not belong to the relation. The applicable corruption operations on a positive example are defined a priori. A list, for example, is corrupted by randomly erasing an element, adding an element, or

switching two consecutive elements. The choice of the corruption operation is also random. The universe of 'near miss' negative examples involving lists of lengths, equal or less than 4, is called Unm4(-). Similarly we can have Unm3:5(-), Unm2i(-), etc.

Random negative examples are simpler to generate than near misses, since they require less processing. In our opinion, however, near misses tend to simulate better the kind of negative examples a real user would give. For that reason we only report here the results obtained with near miss negative examples.

6.1.4 The SKILit parameters

For every experiment it is important to take into account the state of the SKILit parameters.

parameter	**default value**	**meaning**
solver_depth	6	controls the interpreter depth in the coverage tests.
max_num_of_refinement_nodes	300	maximum number of refinements generated during the construction of a clause.
dcg	decomp_test_rec_comp_2	the clause structure grammar used to define the language bias.

Table 6.1: SKILit parameters.

When the parameter values for an experiment are not explicitly mentioned, the default values are assumed.

6.1.5 Predicates used in the experiments

The predicates used for evaluation are some common list processing predicates.

- *member(int,list)*: This predicate is true if the integer in the first argument is contained in the list in the second argument.
- *last_of(int,list)*: The integer is the last element of the list.
- *delete(int,list,list)*: The second list is obtained from the first by removing the first occurrence of the integer from it. If the integer is not in the first list, the predicate fails.
- *rv(list,list)*: The second list has the same elements of the first one in reversed order.
- *append(list,list,list)*: The third list is obtained by concatenating the first list with the second one.
- *split(list,list,list)*: The second list contains the elements which are in odd positions in the first list. The third list contains the elements that are in even positions in the first list.
- *union(list,list,list)*: Each list represents a set and is assumed to have no repeated elements. The third list contains all the elements from the first two lists, without repetitions.

The definitions of all the above predicates are shown in Appendix A.

6.1.6 Overview of the experiments conducted

In the first series of experiments SKILit described in Section 6.2 was evaluated on its own. The negative examples used were near misses. The positive examples used for testing are more complex than the ones used for training. The reason for this is that more demanding test sets (U3:5(+)) reduce the possibility of having a not so good program (in the sense that it would not be accepted by a human programmer) achieve a high success rate. This option was motivated by the results we obtained on some early experiments not reported here (see Appendix D). In those experiments less demanding test sets

(U4(+)) were used and we observed that some programs synthesized by SKILit achieved maximum success rate, despite being clearly imperfect. In Section 6.3 we describe experiments with the synthesis of predicate *union/3* that describe some limitations of the evaluation methodology as well as of the synthesis methodology. In Section 6.4 we give results of experiments comparing SKILit with the systems CRUSTACEAN and Progol. In Section 6.5 we show the results of other experiments conducted with SKILit.

6.2 Results with SKILit

In this series of experiments we evaluate the performance of system SKILit with respect to the number of randomly chosen positive and negative training examples on some relatively simple predicates (*append/3*, *delete/3*, *last_of/2*, *member/2*, *rv/2*, *split/3*). The positive training examples are withdrawn from the universe U4(+) of every relation, and the negative examples from the universe Unm4(-). Regarding the test sets, positive examples were randomly extracted from the universe U3:5(+).

6.2.1 Success rate

Figure 6.2 shows the learning curves for the average success rate obtained by SKILit for each one of the six predicates considered. For each predicate, four curves are presented. One for 0 negative examples, the others for 5, 20 and 100. Every curve shows the average success rate obtained over 10 repetitions for 2, 3, 5, 10 and 20 positive examples.

For five out of the six predicates considered in this experiment, SKILit was able to reach a success rate equal to 1 with 20 positive examples and 100 negative examples. One exception was the predicate *append/3* that remained at a maximum level of 0.85.

Some simpler predicates (*delete/3*, *last_of/2*, *member/2*, *split/3*) reached the maximum success rate with 10 positive examples and 5 negative. The system can obtain good results, in terms of success rate, even in the absence of training negative examples. This

is fundamentally due to the following reason. The various sources of bias used by SKILit (clause structure grammar, background knowledge, parameters, clause construction strategies, etc.) are sufficient to eliminate many definitions that cover test negative examples. This is the case of the induction of predicate *member/2*, for instance, where excellent results are obtained with 10 positive examples and 0 negative ones. For almost all predicates we observe in these experiments little or no variation in the success rate with respect to the number of negative training examples.

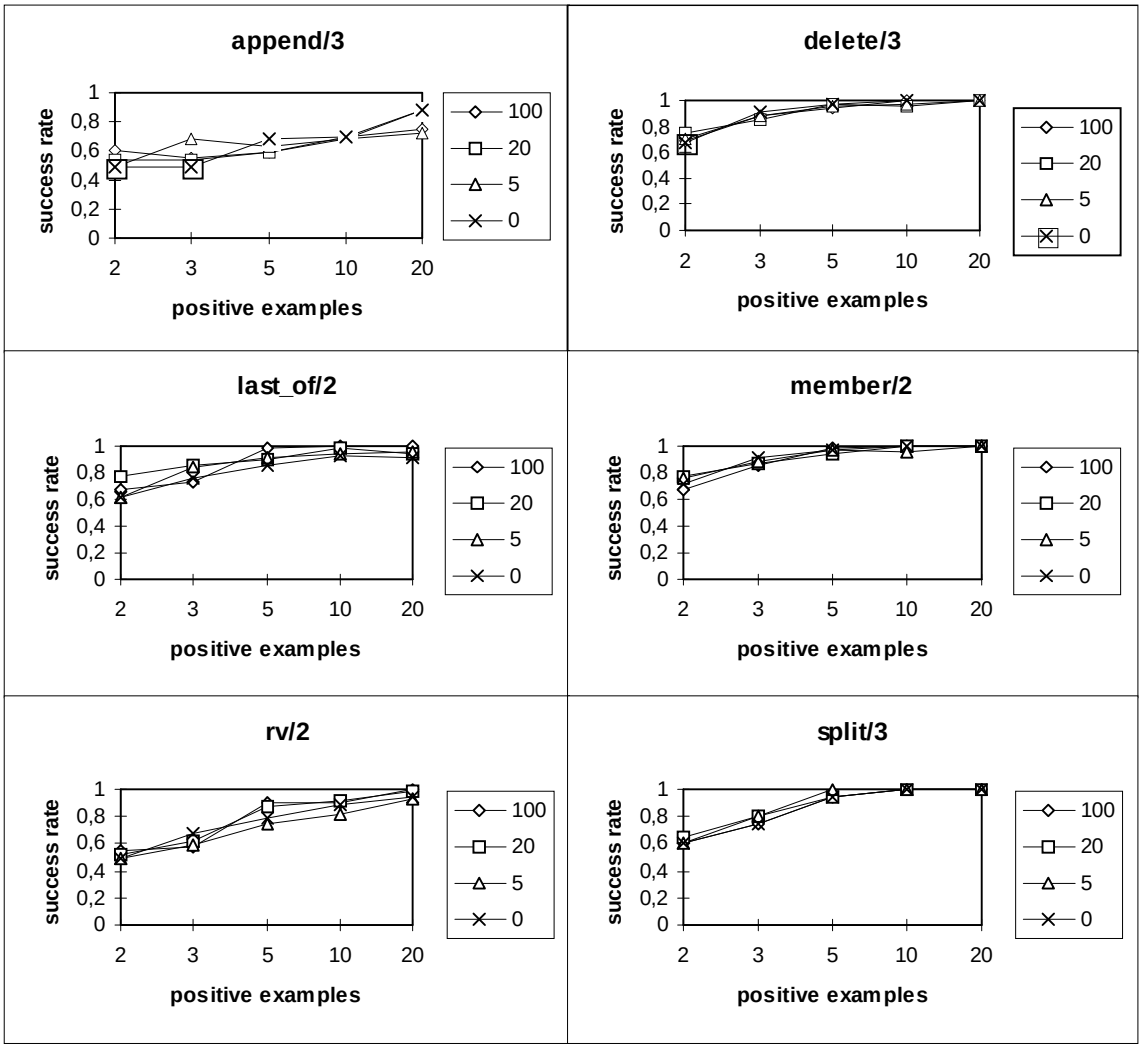

Figure 6.2: Success Rate vs. the number of training examples.

6.2.2 Percentage of test-perfect programs

In Figure 6.3 we show the learning curves for the percentage of test perfect programs.

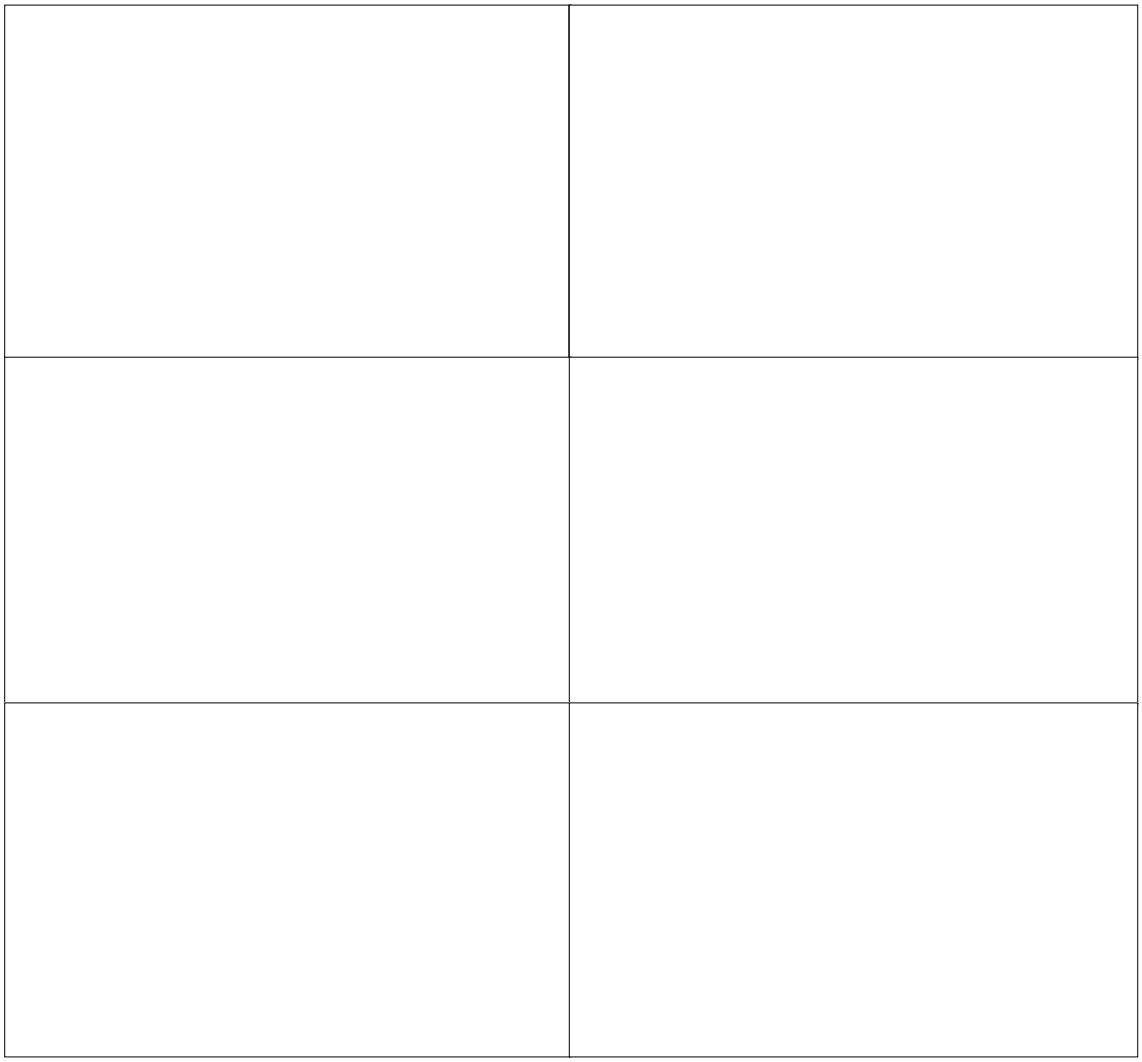

Figure 6.3: Percentage of test-perfect programs vs. the number of training examples.

Contrary to the success rate, the percentage of test-perfect programs varies noticeably with the number of negative examples. This happens, for instance, with the predicates *append/3*, *last_of/2* and *rv/2*. Furthermore, these predicates obtain 0% of test-perfect programs when 0 negative examples are supplied. Once again, the system rapidly converges towards 100% when the number of examples increases. The exception is still the predicate *append/3*.

6.2.3 CPU time

The average CPU time spent for the various experiments is shown in Figure 6.4.

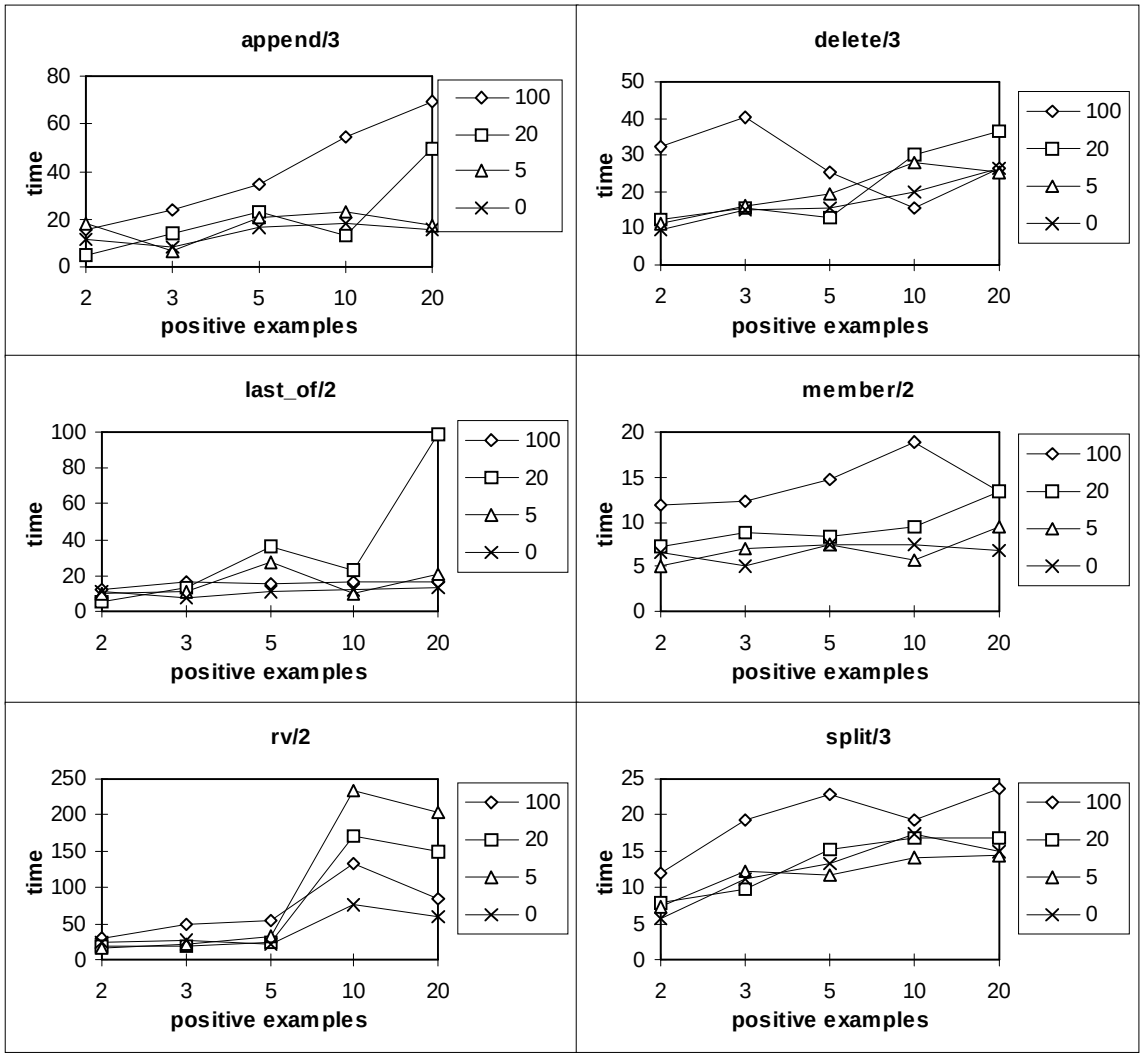

Figure 6.4: Spent CPU time (seconds).

We can observe some irregularity in the time curves relative to the number of examples. However, SKILit behaves well as the number of positive and negative training examples grows. In most cases, the CPU time does not dramatically increase with the number of training examples. The CPU time SKILit spends seems to be more affected by the quality of positive examples rather than their quantity.

6.3 Experiments with union/3

Systematic evaluation following the methodology described before has also been carried out with other predicates. For some of them (*factorial/2*, *extNth/2*, *noneiszero/1*), results were somewhat similar to the ones presented for the six predicates used earlier.

However, for more complex predicates, such as *union/3*, *quicksort/2* (*qsort/2*), *insert/3* and *partition/4*, good positive examples and/or good negative examples are hard to generate randomly in our methodology. These predicates have sometimes more than one recursive clause or more than one base clause, each of which involving one particular test literal. Others have more than one recursive literal.

This does not necessarily mean that our synthesis methodology needs carefully chosen examples for those predicates. It may also be the case that simulating a human user by randomly generating examples is harder for more complex predicates. For example, many of the negative examples randomly generated for predicate *union/3* tend to be variants of a few different cases. On the other hand some important negative examples are unlikely to be randomly generated. We should note that the process of random generation of examples was the same for all the predicates evaluated (from *member/2* to *partition/4*).

We will now describe some experiments conducted with the synthesis of predicate *union/3* which help describing some of these difficulties.

We ran SKILit on sets of 30 randomly generated positive examples and 100 negative examples. Positive examples were taken from the universe U2i(+), and negative examples from the universe Unm2i(-). Test examples were taken from U3:5(+) and Unm2i(-). The number of repetitions per experiment was 50.

With randomly constructed training sets, SKILit did not synthesize one test perfect program in 50 runs (Table 6.2). We can improve the results by changing the clause structure grammar so that every clause is forced to have at least one test literal (the

previous clause structure grammar allowed clauses without test literals). The new CSG is called decomp_+test_rec_comp_2. In 50 runs, SKILit finds 3 test-perfect programs.

Positive	*Negative*	*CSG*	*success*	*test-perfect*	*time*
random	near misses	decomp_test_rec_comp_2	0.532	0	570.484
random	near misses	decomp_+test_rec_comp_2	0.585	6	543.441
random	9 chosen	decomp_test_rec_comp_2	0.753	34	204.748
random	9 chosen	decomp_+test_rec_comp_2	0.740	34	130.772
5 chosen	near misses	decomp_test_rec_comp_2	0.794	58	63.106
5 chosen	near misses	decomp_+test_rec_comp_2	0.821	64	50.838

Table 6.2: Experimental results for *union/3*.

Are these bad results due to the lack of good positive examples or good negative examples? To answer this question we ran SKILit again with 50 sets of 30 random positive examples. However this time 9 negative examples were manually chosen. The results improved clearly independently of the grammar used. For the clause structure grammar decomp_+test_rec_comp_2, however, the CPU time spent was considerably less (Table 6.2). The chosen positive and negative examples are shown in Table 6.3.

Chosen positive examples	*Chosen negative examples*
union([],[2,3],[2,3]). *union([2],[2,3],[2,3]).* *union([2],[3,4],[2,3,4]).* *union([2,3],[4,2,5],[3,4,2,5]).* *union([2,3],[4,5],[2,3,4,5]).*	*-union([2],[1,2],[2,1,2]).* *-union([2],[3,4],[3,4]).* *-union([3],[2],[3]).* *-union([2,3],[2],[2]).* *-union([2,3],[4],[3,4]).* *-union([2,3],[4],[2,4]).* *-union([2],[2],[2,2]).* *-union([2,1],[2],[2,1,2]).* *-union([1,2],[1,2],[1,1,2]).*

Table 6.3: Chosen positive and negative examples used in the experiments.

We then picked 5 chosen positive examples (Table 6.3) and ran SKILit with 50 sets of 100 random negative examples. The results obtained are quite good with the usual clause structure grammar and still improve if we use the grammar that imposes test literals. We can conclude that it is likely (34%) to find good random sets of positive examples when good negative ones are manually chosen. It is also likely (58%, 64%) to find good

random sets of negative examples when good positive ones are given. However, finding two good random sets simultaneously has low probability (0%, 6%) (Table 6.2).

One possible direction is to give the user more powerful means to transmit negative examples to the system. This motivated our work with integrity constraints, presented in the next Chapter.

6.4 Comparison with other systems

Here we concentrate on comparing SKILit with CRUSTACEAN and Progol, since these two systems seem to be representative of the state-of-the-art. Nevertheless, other previously described works, are also relevant.

6.4.1 CRUSTACEAN

A comparison between the SKILit system and the CRUSTACEAN system, conducted on some predicates whose results appear in [53], is summarised in Table 6.4 and shown graphically in Figure 6.5. The values shown for CRUSTACEAN were taken from [1]. The values for SKILit were obtained from experiments performed by us in conditions, as identical as possible, to those described by Aha et al. For this reason we should consider these values only as indicative. Anyhow, in the presence of a rather reduced number of positive and negative examples, the SKILit system obtains success rate results comparable to the CRUSTACEAN system. Taking into account that SKILit uses a much weaker language bias than CRUSTACEAN (which implies a larger search space and a wider applicability) it is an important result (see Section 3.5.4).

	SKILit			*CRUSTACEAN*	
	number of training positive examples				
	2	3	5	2	3
append/3	0.76	0.80	0.89	0.63	0.74
delete/3	0.75	0.88	1.00	0.62	0.71
rv/2	0.66	0.85	0.87	0.80	0.86
member/2	0.70	0.89	0.95	0.65	0.76
last_of/2	0.71	0.72	0.94	0.74	0.89

Table 6.4: SKILit's success rate vs. CRUSTACEAN's

For each predicate we varied the number of positive examples between 2 and 5, whilst the number of negative examples was kept constant (=10). The positive examples were extracted from the universe U4(+). For the negative examples we used the universe Unm4(+). The results show averages obtained over 5 runs.

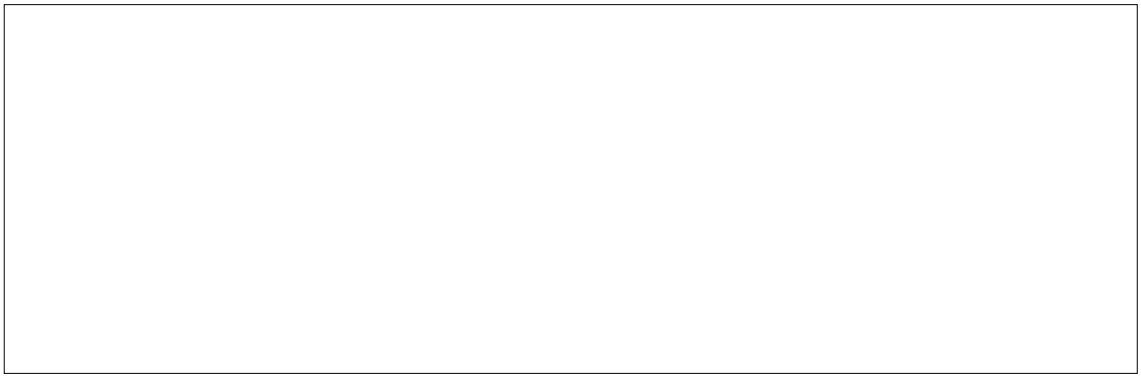

Figure 6.5: Success rates of SKILit vs. CRUSTACEAN

6.4.2 Progol

To compare SKILit with the system Progol [80], we decided to take one of the input files distributed with Progol itself. This file contains 17 positive examples and 8 negative ones for predicate *append/3*. With these examples, the Progol version that was available to us, perfectly synthesizes a definition of *append/3*. The comparison experiment with SKILit consisted in running both systems over 20 sub-sets of those 17 positive examples. These subsets were randomly constructed and supplied to each system, together with all the negative examples. The results are shown in Table 6.5.

	Success Rate		Standard Deviation		% Test-perfects	
	SKILit	*Progol*	*SKILit*	*Progol*	*SKILit*	*Progol*
3	0.625	0.500	0.217	0.000	25	0
5	0.750	0.525	0.250	0.109	50	5
7	0.800	0.699	0.245	0.244	60	35
9	0.900	0.949	0.200	0.150	80	85
11	0.975	0.945	0.109	0.149	95	60
13	0.850	0.996	0.229	0.006	70	70
15	0.975	0.998	0.109	0.006	95	90
17	1.000	0.998	0.000	0.006	100	90

Table 6.5: Comparison between SKILit and Progol for predicate *append/3*.
(The first column shows the number of positive examples)

The same results are represented graphically in Figure 6.6.

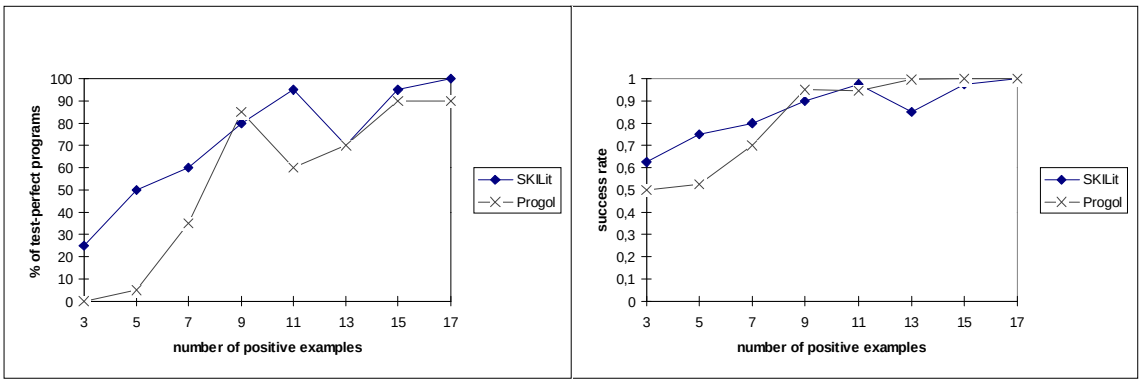

Figure 6.6: Comparison between SKILit and Progol for *append/3*.

Comparing the percentages of the test-perfect programs obtained by each system, we observe that SKILit gets better results. Taking into account that we are in a program synthesis context, this is a very positive result. With respect to the success rate, SKILit seems to achieve better results when tackling very small sets of examples (3 to 7 examples).

6.5 Other Experiments

In this Section we show some isolated experiments conducted with SKILit with manually selected positive and negative examples. These experiments serve to illustrate the class of programs SKILit is able to induce.

Note that in all these experiments the same clause structure grammar has been used (decomp_test_rec_comp_2), unless stated otherwise. This is the same grammar used in all the other experiments reported in this thesis.

The background knowledge is either 'integer' or 'list' depending on the admissible predicates. Unfortunately, a list of the admissible predicates has to be given to SKILit. This is an important limitation that we must investigate in the future.

6.5.1 Factorial

Predicate *factorial(X,Y)* is true if $Y = X!$.

Input:

mode(factorial(+,-)).
factorial(2,2).
factorial(3,6).
factorial(4,24).

-factorial(3,3).
-factorial(4,12).

background_knowledge(integer).
adm_predicates(factorial/2,[succ/2,pred/2,zero/1,multb/3,factorial/2]).
clause_structure(decomp_test_rec_comp_2).

Output:

factorial(A,A)←pred(A,B),pred(B,C),zero(C).
factorial(A,B)←pred(A,C),factorial(C,D),multb(D,A,B).

Number of iterations: 3
507 refinements (total)

6.5.2 Multiply

Predicate *multiply(A,B,C)* instantiates *C* with *A*B*.

Input:

mode(multiply(+,+,-)).
type(multiply(int,int,int)).

multiply(0,4,0).
multiply(1,5,5).
multiply(2,3,6).
multiply(3,4,12).

-multiply(2,2,2).
-multiply(3,3,6).
-multiply(2,2,5).
-multiply(2,2,6).
-multiply(2,2,8).
-multiply(2,3,4).

background_knowledge(integer).
adm_predicates(multiply/3,[pred/2,succ/2,zero/1,plus/3,one/1,multiply/3]).
clause_structure(decomp_test_rec_comp_2).

Output:

multiply(A,B,A)←zero(A).
multiply(A,B,C)←pred(A,D), multiply(D,B,E), plus(E,B,C).

Number of iterations: 3
795 refinements (total)

Properties generated (discarded by TC):

multiply(1,A,A).

6.5.3 Insert

Predicate *insert(I,L1,L2)* inserts an integer *I* into a sorted list *L1* obtaining *L2* so that *L2* is sorted.

Input:

mode(insert(+,+,-)).
type(insert(int,list,list)).

-insert(2,[1],[2,1]).
-insert(1,[2],[2,1]).

-insert(3,[1,2],[3,1,2]).
-insert(3,[1,2],[1,3,2]).

insert(2,[],[2]).
insert(1,[2],[1,2]).
insert(2,[1],[1,2]).
insert(3,[1,5],[1,3,5]).

background_knowledge(list).
adm_predicates(insert/3,[dest/3,const/3,'</2,null/1,insert/3]).
clause_structure(decomp_test_rec_comp_2).

Output:

insert(A,[],[A]).
insert(A,[B|C],[A,B|C])←A<B.
insert(A,[B|C],D)←insert(A,C,E),insert(B,E,D).

This definition of *insert/3* seems logically correct. Computationally it has the following problem. Posing to the program the query

←insert(8,[1,3,5,7],R).

we get a series of identical answers

R = [1,3,5,7,8] ;
R = [1,3,5,7,8] ;
R = [1,3,5,7,8] ;
R = [1,3,5,7,8] ;
R = [1,3,5,7,8] ;
R = [1,3,5,7,8]
....

This kind of computational problem could be easily detected by limiting the number of identical answers obtained on the training examples. This filter, however, is not currently implemented in SKILit.

If we change the CSG so that double recursion is not allowed (decomp_test_rec1_comp_2, Appendix C) we get the following definition for *insert/3*:

insert(A,[],[A]).

insert(A,[B|C],[A,B|C])←A<B.
insert(A,[B|C],[B|D])←B<A,insert(A,C,D).

6.5.4 Partition

The predicate *partition(I,L,S,G)* splits list *L* into two lists *S* and *G*, so that *S* contains the elements of *L* smaller or equal to *I*, and *G* contains the elements of *L* greater than *I*.

Input:

mode(partition(+,+,-,-)).
mode(partition(int,list,list,list)).

partition(2,[],[],[]).
partition(4,[2,6],[2],[6]).
partition(3,[1,2,5],[1,2],[5]).
partition(3,[6,2,5],[2],[6,5]).

-partition(2,[2],[2],[2]).
-partition(2,[1,3,4],[1,4],[3,4]).
-partition(2,[3,1],[3],[1]).
-partition(2,[1,0],[1],[0]).
-partition(2,[4,5],[4],[5]).
-partition(4,[1,2],[1],[2]).
-partition(2,[3],[3],[]).
-partition(2,[1],[],[1]).

background_knowledge(list).
adm_predicates(partition/4,[dest/3,const/3,null/1,'<'/2,partition/4]).
clause_structure(decomp_test_rec_comp_2).

max_num_of_refinement_nodes (1000).

Output:

partition(A,[],[],[]).
partition(A,[B|C],[B|D],E)←B<A, partition(A,C,D,E).
partition(A,[B|C],D,[B|E])←A<B,partition(A,C,D,E).

Properties generated:

partition(A,[B,C],[B],[C])←B<A,A<C.

6.5.5 Insertion sort

Predicate *isort(A,B)* sorts list *A*. List *B* is the sorted list. The admissible predicates chosen (*insertb/3*) determine the sorting strategy.

Input:

mode(isort(+,-)).
type(isort(list,list)).

isort([],[]).
isort([2,1],[1,2]).
isort([3,2,1],[1,2,3]).

-isort([1,2],[2,1]).
-isort([1,3,2],[1,3,2]).
-isort([3,2,1],[2,3,1]).
-isort([3,2,4,1],[2,3,4,1]).

background_knowledge(list).
adm_predicates(isort/2,[dest/3,const/3,insertb/3,isort/2,'<'/2,null/1]).
clause_structure(decomp_test_rec_comp_2).

Output:

isort([],[]).
isort([A|B],C)←isort(B,D),insertb(A,D,C).

Number of iterations: 3
193 refinements (total)

Properties generated:

isort([A,B],C)←insertb(A,[B],C).

6.6 Related work concerning evaluation

Some ILP systems have been empirically evaluated in a systematic way by their authors. Next we describe the most relevant pieces of work here.

In 1993 Quinlan presented his "*Midterm Report*" where system FOIL is evaluated [97] using an experimental methodology considered to be "more pragmatic" than the usually used for machine learning/ILP systems. Quinlan criticized in particular the fact that many systems have been evaluated using a very limited background knowledge, as well as a set of carefully chosen training examples. The evaluation methodology adopted by Quinlan consists of making the system synthesize a series of list processing predicates from

Bratko's book, "*Prolog Programming for Artificial Intelligence*" [10]. The background knowledge is initially empty and accumulates the predicates as it learns.

With respect to the training examples, Quinlan considers the set of lists with length three or less, and with elements from the set {1,2,3}. This set of lists is called universe U3. Another set, universe U4, has lists of length four or less and elements in {1,2,3,4}. For each relation and for each universe he constructed the set of positive examples belonging to the relation involving the lists in that universe. Those that do not belong to the relation constitute the set of negative examples. All the positive examples are assigned for training, as well as all the negative examples up to the limit of 90 000 examples. The results obtained are considered satisfactory for most of the predicates.

In relation to Quinlan's work, we would like to point out the following. In spite of the fact that the training sets are not carefully chosen, they are complete for each universe considered (U3, U4). This fact facilitates the induction of recursive definitions. In an inductive synthesis setting, where the user manually supplies the examples, one cannot expect the training sets to be complete. Instead, the training sets tend to be small and sparse.

In 1994 Aha et al. [1] evaluated the CRUSTACEAN system using randomly selected training sets, in order to demonstrate that the system can synthesize recursive definitions from a small (and sparse) set of examples. For each predicate, CRUSTACEAN's performance (success rate) was measured with training sets of 2 to 3 positive examples and 10 negative examples, all of which were randomly chosen. The performance values are obtained from an average of 10 runs for a given number of positive examples. The predicates involved are *append/3*, *delete/3*, *extractNth/3*, *factorial/2*, *last_of/2*, *member/2*, *noneIsZero/1*, *plus/3*, *reverse/2* and *split/3*.

Our experimental methodology is largely based on this work. Their evaluation already confronts an induction system to situations where the training sets are not complete, but sparse. The methodology's main limitation is, in our view, its difficult application to

systems with a wider set of synthesizable programs than CRUSTACEAN. A system like SKILit, for example, which can induce a great variety of clauses, requires negative examples that can eliminate over-general programs. It is unlikely that these negative examples are randomly generated in the methodology of Aha et al.

7. Integrity Constraints

In this Chapter we introduce the integrity checking algorithm MONIC which uses a Monte Carlo strategy to search for inconsistencies between a program and integrity constraints. MONIC is efficient, sound, but incomplete. MONIC is integrated with the SKILit system and we present experimental results regarding the synthesis of programs from positive examples and integrity constraints.

7.1 Introduction

Typical ILP systems accept ground negative examples only. In our view, a ground atom conveys very little information when it is given as a negative example. The practical result of this is that the number of negative examples given to a system tends to be high. The fact that ILP systems usually require a large number of negative examples is the main problem tackled in this Chapter.

To illustrate this problem we present two examples referred to in the literature.

- According to the "*mid-term report*" by Quinlan [97] the FOIL system learns the predicate *reverse/2* using 1560 to 92796 negative examples.

- Zelle et al. [125] refers that the CHILLIN system learns the predicate *member/2* with an average accuracy of around 50% given more than 80 negative examples.

These facts restrict the applicability of ILP systems, especially when the examples are manually supplied by the user, as it happens in program synthesis.

The problem of the excessive number of negative examples has already drawn some attention from the ILP community. Some systems like FORCE2 [12], LOPSTER [60] and CRUSTACEAN [1] use a very restrictive language bias, which reduces the number of negative examples required. However, these systems seem difficult to extend to cope with a greater variety of logic programs. The FOIL system [96] allows the use of a *closed world assumption*. If a fact is not given to the system explicitly as a positive example, it is then considered a negative example. This technique is not practical in many learning situations because it forces the user to supply a set of complete positive examples.

A promising alternative relies on the use of *integrity constraints*. These are first order logic clauses of the form $a \wedge \ldots \wedge b \rightarrow c \vee \ldots \vee d$, that can be used to transmit to the system some conditions that the predicate to synthesize should respect, as it happens with negative examples. The main difference is that the integrity constraints enable a more compact representation than atomic negative examples. Luc de Raedt suggested that an ILP system could use constraints to verify the generated programs [21].

In spite of the fact that integrity constraints are not normal program clauses, the *SLDNF proof procedure* can be used to check whether a logic program satisfies an integrity constraint, by transforming the constraint into a *query*, and posing that query to the logic program. This strategy, however, suffers from severe efficiency problems, since finding a violating instance of the constraint may involve trying all its possible instantiations. Other

more sophisticated special integrity constraint handlers, like SATCHMO [65], still seem computationally too heavy for practical use in ILP.

We propose a new method to handle integrity constraints. It enables the use of constraints in ILP systems without high efficiency costs. Experimental results show that we can induce quite accurate recursive logic programs rather efficiently this way. In fact, in our experiments, we observed that for the same level of accuracy, our system runs faster with integrity constraints than with negative examples.

Our integrity constraint checker (MONIC) uses a 'Monte Carlo' strategy. To check whether a program *P* satisfies an integrity constraint *I*, it randomly generates a number of logical consequences of *P* and verifies if they satisfy *I*. This is a very efficient way to handle constraints. Unfortunately our constraint checker is also incomplete. However, we can control the level of incompleteness by varying the number of logical consequences sampled from *P*.

7.2 The number of negative examples

As it was already referred in Chapter 5, many ILP systems require an excessive number of positive examples to induce predicate definitions. This a serious problem and a barrier to the usability of ILP systems, especially in the program synthesis context. System SKILit handles the lack of crucial training positive examples by generating properties. These are clauses that capture regularities within the positive examples, generalize them, and enable the introduction of recursive clauses.

What about negative examples? Giving all the crucial negative examples to an ILP system can also be tedious, as there can be a large number of them. One ground negative example conveys little information to the system. Besides, the user does not know which negative examples are more appropriate for the synthesis task. Thus she/he tends to give the system more negative examples than necessary.

7.3 Integrity constraints

A property can represent a set of positive examples. Likewise, the negative examples can also be replaced by, or complemented with, more expressive clauses. Such clauses are called *integrity constraints*.

Example 7.1: We can express that no term is member of the empty list through the integrity constraint $member([],X) \rightarrow false$. ♦

Example 7.2: The clause $sort(X,Y) \rightarrow sorted(Y)$, represents an integrity constraint which expresses the condition "the second argument of predicate *sort/2* is a sorted list". Likewise, we can say that the list Y is a permutation of list X with $sort(X,Y) \rightarrow permutation(X,Y)$. ♦

Integrity constraints, like negative examples, can be used by an ILP system to detect or reject over-general programs. In fact, the negative examples can be seen as a special case of integrity constraints. For example, $member([],2) \rightarrow false$ represents the negative example $member([],2)$. An integrity constraint intensionally represents a possibly infinite set of negative examples and can express the negative information in shorter terms than ground negative examples.

Definition 7.1: An integrity constraint is a first order clause of the form $A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m$. The A_i and the B_i are atoms. The A_i are called positive conditions and the B_i, negative conditions of the constraint. ♦

Note that $A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m$ can be written as $B_1 \wedge \ldots \wedge B_m \rightarrow A_1 \vee \ldots \vee A_n$. Here, we will adopt a Prolog-like notation as we did for other clauses[6]. The disjunction and conjunction operators are replaced by commas as in $B_1,\ldots,B_m \rightarrow A_1,\ldots,A_n$. The commas on the antecedent side represent conjunctions. Those on the consequent side represent

disjunctions. As we did for program clauses, we will keep the arrow (→). Negation is interpreted as negation as failure.

Example 7.3: The integrity constraint

union(A,B,C),member(X,C)→member(X,A),member(X,B)

expresses the condition that if *X* belongs to the list in the third argument of *union/3* (output argument) then it is either a member of the list in the first argument or of the list on the second one (one of the input arguments). ♦

Integrity constraints are generally defined as *range restricted* clauses [21,105]. In this work we do not consider this restriction, since the programs we synthesize are not range restricted either.

Positive and negative examples can also be represented as integrity constraints. A positive example *p* corresponds to the constraint *true→p*. A negative example *n* is represented as the constraint *n→false*. Although examples and constraints can theoretically be handled in a uniform way, we do it separately since we use different strategies to handle positive examples, negative examples and integrity constraints.

7.3.1 Constraint satisfaction

In order to avoid the induction of over-general programs, an ILP system should test the candidate program against the integrity constraints on certain occasions. If a candidate program *P* satisfies the integrity constraints, then it is accepted and the induction process proceeds to the next phase.

---

[6] Obviously, constraints are more expressive than Prolog Clauses since the first can have more than one literal in the head.

We now define the notions of *satisfaction*, *violation* and *violating instance* of an integrity constraint [21]. The agent that verifies the constraint satisfaction will be called *integrity constraint checker* or simply *integrity checker*.

Definition 7.2: Given a constraint $B_1,\ldots,B_m \rightarrow A_1,\ldots,A_n$ and a program P, the constraint is *satisfied* by P if and only if the query $\leftarrow B_1,\ldots,B_m$,not $A_1,\ldots$,not A_n fails on P. If P does not satisfy I we say that P *violates* the constraint I. If IT is a set of integrity constraints, P satisfies IT if it satisfies all the constraints in IT. ♦

Definition 7.3: If I is an integrity constraint $B_1,\ldots,B_m \rightarrow A_1,\ldots,A_n$, $I\theta$ is a *violating instance* of I if and only if θ is a possible answer substitution for the query $\leftarrow B_1,\ldots,B_m$,not $A_1,\ldots$,not A_n, when presented to P. ♦

Example 7.4: The integrity constraint *sort(X,Y)→sorted(Y)* is not satisfied by program {*sort(X,X)*} ∪ {definition of *sorted/1*}. We can check that by transforming the constraint into the query

←*sort(X,Y)*,not *sorted(Y)*.

This query succeeds on {*sort(X,X)*} ∪ {definition of *sorted/1*} with the answer substitution {*X/[1,0]*, *Y/[1,0]*}. Thus, a violating instance is

sort([1,0],[1,0])→sorted([1,0]).

♦

Given Definition 7.2, we can check whether a program P satisfies a constraint by transforming the constraint into a query and posing that query to P, using SLDNF. Although this is a simple way to check consistency, it is potentially inefficient. It is convenient because it does not require special theorem provers. Its inefficiency is due to the generate-and-test nature of SLD(NF).

Example 7.5: Integrity constraint *sort(X,Y)→sorted(Y)* can be transformed into a query ←*sort(X,Y)*,not *sorted(Y)*. To check the consistency of the constraint and a program *P*, we pose the query to *P*. SLDNF constructs all possible instantiations of the literal *sort(X,Y)* and, for each value of *Y*, it tests whether or not it is a sorted list (assuming *X* and *Y* range over lists). When an unsorted list is found, we have a violating instance of the constraint. This can be very inefficient. Suppose we are considering that *X* and *Y* range over lists of length 0,1,2,3 and 4, with integer elements from {0,1,…,9}. This represents a universe of more than 10000 lists. To answer such a query, SLDNF may have to try all possible values. This problem gets exponentially hard as the arity of the predicate in the first literal increases. ♦

Relatively little attention has been given to integrity constraints in the field of machine learning, inductive logic programming included. Luc De Raedt employed integrity constraints in his system CLINT [21]. The constraints were transformed into queries and confronted with the induced programs, as suggested by Definition 7.2. For this reason, the search for a violating instance is inefficient. If any violating instance is found, CLINT attempts to determine which predicate is incorrectly defined with the help of an oracle.

One can find other integrity checkers in the logic programming literature, such as SATCHMO [65], and the one by Sadri and Kowalski [105]. The problem with this sort of integrity checkers is their inefficiency.

7.4 MONIC and the Monte Carlo strategy

In this Section we describe MONIC (*Monte Carlo Integrity Checker*), a *Monte Carlo method*[7] [103] which handles integrity constraints. The method is incorporated within

[7] According to Rubinstein [103], the designation “Monte Carlo” was introduced by von Neumann and Ulam during the World War II, as code for the secret work being carried out at Los Alamos, while the Method of Monte Carlo was applied to problems related to the atomic bomb. Nowadays, still according to Rubinstein, it is still the most powerful and the most used method in complex simulation problems with a broad scope of application.

system SKILit. As seen in Chapter 5 SKILit constructs a logic program P by adding one clause C at a time to an initial theory P_0. Algorithm 6 describes the induction process and shows where the integrity check is made.

$P := P_0$
while P does not satisfy some stopping criterion
    construct new clause C
    if $P \cup \{C\} \cup BK$ satisfies integrity constraints
        $P := P \cup \{C\}$
    end if
end while

Algorithm 6: High level description of SKILit

In each cycle, after the generation of a clause C, there is a consistency check which involves the new program $P \cup \{C\}$. This new program is accepted only if it satisfies the integrity constraints. We now describe how the integrity constraints are processed.

7.4.1 Operational integrity constraints

MONIC processes integrity constraints of the form $A_1,\ldots,A_n \rightarrow B_1,\ldots,B_m$, as defined earlier. Furthermore, two conditions are imposed to an integrity constraint I given in the specification of a program P defining a predicate p/k.

1. The leftmost literal of the antecedent, should be a positive literal with predicate p/k.

2. If I is transformed into query Q of the form $\leftarrow B_1,\ldots,B_m$,not $A_1,\ldots,$not A_n and the input arguments of B_1 are instantiated, then the query should be an *acceptable* query with respect to the input/output modes of the predicates in Q.

A query $\leftarrow L$, where L is either $p(X_1,\ldots,X_n)$ or *not* $p(X_1,\ldots,X_n)$, is acceptable if every input argument X_i is fully instantiated. A query $\leftarrow p(X_1,\ldots,X_n)$, *MoreLiterals* is *acceptable* if, after instantiation of all arguments X_i, the query $\leftarrow$*MoreLiterals* is acceptable, where *MoreLiterals* is a conjunction of literals. A query $\leftarrow$*not* $p(X_1,\ldots,X_n)$,

MoreLiterals is *acceptable* if the query ←*MoreLiterals* is acceptable. Checking the acceptability of a query is trivial given the input/output mode declarations of the involved predicates. This condition guarantees that the input/output mode of the involved predicates will be respected.

The first condition guarantees that the integrity constraint restricts predicate *p/k*, since the literal with the predicate *p/k* is found in the antecedent of the constraint. The fact that this literal must be in the leftmost position allows the search for a violating instance of the integrity constraint to start from a ground logical consequence of program *P*.

The integrity constraints accepted by MONIC are *restrictive* in the sense defined by De Raedt [21]. In this type of restrictions, the literals relative to the predicate to be induced are in the antecedent. An example of a restrictive integrity constraint relative to predicate *union/3* is

union(A,B,C),member(X,A)→member(X,C).

This constraint says that if list *C* is the union of lists *A* and *B*, then every element *X* of *A* should be an element of *C*.

An integrity constraint which has the predicate to be induced in the consequent is called *generative*. An example of a generative constraint relatively to predicate *union/3* is *true→union(A,A,A)*. Here we do not consider this sort of constraints, although it seems possible to extend our integrity checker to handle them.

7.4.2 The algorithm for constraint checking

Our consistency checking algorithm (MONIC) takes a particular program *P* defining some predicate *p/k* and a set of integrity constraint *IT*, and gives one of two possible answers. Either *P* and *IT* are *inconsistent* (some $I \in IT$ is violated), or *P* and *IT* are not found to be inconsistent, and are considered *probably consistent*.

The Monte Carlo method is based on the random generation of facts concerning the predicate *p/k* which are logical consequences of the program *P*. Each of these facts is used to search for an instance of some $I \in IT$ which is logically false. If such a violating instance is found, we can be sure that *P* and *IT* are inconsistent. if no violating instance of some $I \in IT$ is found, after a limited number of attempts we stop. In that case, *P* and *IT* are not found to be inconsistent, but only probably consistent.

The random generation of ground logical consequences of *P* is central to the algorithm, and deserves some more attention. To obtain fact *f*, such that $P \vdash f$, we start with the most general term $p(X_1,\ldots,X_k)$ of *p/k* ($X_1,\ldots X_k$ are variables). For clarity, we assume k=2 in the following. Let us also suppose *mode(p(+,-))*, and $type(p(type_X,type_Y))$, for the most general term *p(X,Y)*. We now want a query $\leftarrow p(X,Y)$, where *X* is bound to a term of type $type_X$ (remember that *X* is an input argument). For that, *X* unifies with a term t_{in} of type $type_X$. After querying program *P*, variable *Y* is bound to a term t_{out}. Fact *f* is $p(t_{in},t_{out})$.

The random nature of *f* comes from the choice of the input arguments. Each term of a given type is sampled from a given population of terms with a fixed distribution (see Section 7.4.3).

Given a fact, we unify it with the leftmost literal in the antecedent of each $I \in IT$. The constraint can now be transformed into an acceptable query. The query posed to *P* either succeeds or fails. Success means that a violating instance of *I* was found, and so *P* and *I* are inconsistent. Failure means that, although no violating instance was found, *P* and *I* can still be inconsistent. However, the more facts fail to violate *I*, the more likely it is that *P* and *I* are consistent. Further on, in Section 7.7.1, a probabilistic measure of this likeliness is given.

input: Program *P* defining the predicate *p/k*;
Mode and type declarations of predicate *p/k*;
A set of integrity constraints *IT*;
Integer *n*.

output: One of {inconsistent, probably consistent}

1. Generate query Q
   $p(X,Y)$ is the most general term of p/k
   (X represents the input arguments, Y the output ones)
   For each variable $V_i \in X$, randomly instantiate it with t_i of type $type(V_i)$;
   $\theta_{in} = \{V_i/t_i\}$
   $Q := \leftarrow P(X,Y)\theta_{in}$
2. Pose query Q to P
   If Q fails then return to step 1
   Else we obtain an answer substitution θ_{out}
   (if there are alternative answer substitutions, we consider each one of them)
3. Generate fact f
   $f = P(X,Y)\theta_{in}\theta_{out}$
4. For each $I \in IT$, search a violating instance of I.
   Transform I into a query $\leftarrow L, MoreLiterals$
   θ_{uni} is the unifier of the leftmost literal L and f
   Pose query $\leftarrow MoreLiterals\theta_{uni}$ to P
   If the query succeeds then P violates I
   Store f as a negative example
   Return 'inconsistent'
5. After n queries return 'probably consistent'
   Otherwise return to step 1.

Algorithm 7: MONIC: The integrity checker.

Example 7.6: The program P below, contains an incorrect definition of *rv/2* which is supposed to reverse the order of the elements of a given list. Definitions for *append/3* and *last_of/2* are also given as part of the background knowledge.

```
mode(rv(+,-)).
type(rv(list,list)).
rv([A,B|C],[B,A|C]).
rv([A|B],C)←rv(B,D),append(D,[A],C).

mode(append(+,+,-)).
type(append(list,list,list)).
append([],A,A).
append([A|B],C,[A|D])←append(B,C,D).

mode(last_of(+,-)).
```

type(last_of(list,int)).
last_of([X],X).
last_of([X|Y],Z)←last_of(Y,Z).

The following integrity constraint *I* imposes that for every fact *rv(X,Y)*, the first element of list *X* is the last element of list *Y*.

rv(X,Y), X=[A|B] → *last_of(Y,A).*

We will now follow one iteration of the constraint checker MONIC (Algorithm 7).

Step 1: *rv(X,Y)* is the most general term;
X is the only input argument and has type *list*;
A random choice of a term of type *list* gives *t=[4,1,5]*;
The query *Q* is ←*rv([4,1,5],X)*.

Step 2: The query *Q* succeeds on *P* and we obtain θ_{out}={*X/[1,4,5]*}.

Step 3: *f* is *rv([4,1,5],[1,4,5])*.

Step 4: *I* is turned into the query
←*rv([4,1,5],[1,4,5]), [4,1,5]=[4|[1,5]],* not *last_of([4,1,5],4)*.
The query succeeds. *I* is violated.
Store *rv([4,1,5],[1,4,5])* as a negative example.
Return 'inconsistent'. ♦

7.4.3 Types and distributions

Random facts are obtained by randomly generating the input arguments of a query which is placed upon a program *P*. The random generation of each argument is made according to a distribution defined for the type of that argument. Here, a distribution is associated to every type. The distribution can be pre-defined in the ILP system, or it can be defined by the user himself. Presently, we define the distribution of a type by specifying the probability of obtaining terms of length 0, 1, 2, etc. An alternative for the definitions of type distribution are the stochastic logic programs by Muggleton [81].

Example 7.7: In our experiments we defined the distribution for the set of lists of length 0 to 4, with elements 0 to 9, as follows.

Probability(length of list L=n) = 0.2 for n=0,…,4.

Probability(any element of a list L is d)=0.1 for d=0,…,9.

As a consequence, 0.2 is the probability of obtaining an empty list (*[]*). The probability of obtaining the list *[3]* is 0.2×0.1=0.02. ♦

Although MONIC requires that a particular distribution must be defined for each type, the choice of the distributions does not seem difficult. In fact, the distributions we used in the experiments were practically our first choice.

7.5 Evaluation

We conducted some experiments to evaluate SKILit's performance when combined with the MONIC module for integrity constraint checking. In the first experiments (Section 7.5.1) we chose the predicates *append/3* and *rv/2* and for each one of them tried different sets of integrity constraints. The evaluation methodology is identical to that described in Section 6.2, except that here the integrity constraints are also given. In Section 7.5.2 we describe experiments with the predicate *union/3*. In all the experiments the number of queries generated by Algorithm 7 (Integer n) was 100.

7.5.1 append/3 and rv/2

For the predicate *append/3* the following four sets of integrity constraints were used (see Appendix A for the definitions of *member/2* and *sublist/2*):

ic1: *append(X,Y,Z),member(A,X)→member(A,Z).*
append(X,Y,Z),member(A,Y)→member(A,Z).

ic2: *append(X,Y,Z),sublist([A,B],X)→sublist([A,B],Z).*

append(X,Y,Z),sublist([A,B],Y)→sublist([A,B],Z).

ic3: *append(X,Y,Z),sublist(A,X)→sublist(A,Z).*
append(X,Y,Z),sublist(A,Y)→sublist(A,Z).

ic4: *append(X,Y,Z),sublist(A,X)→sublist(A,Z).*
append(X,Y,Z),sublist(A,Y)→sublist(A,Z).
append([_|_],X,X)→false.

We now explain in words the meaning of some of the constraints. The first constraint in ic1, for instance, says that if list *Z* is the result of appending lists *X* and *Y*, then any element *A* of *X* should be an element of *Z*. The second constraint in ic2 says that if *A* and *B* are two consecutive elements of list *Y*, they should be two consecutive elements of list *Z*. The third constraint of ic4 says that if we append a list with at least one element (*[_|_]*) to list *X*, we should never obtain the same list *X*.

For *rv/2* we have two sets of constraints

ic1: *rv(X,Y),sublist([A,B],X)→sublist([B,A],Y).*

ic2: *rv(X,Y),length(X,N)→length(Y,N).*
rv(X,Y),member(A,X)→member(A,Y).
rv(X,Y),member(A,Y)→member(A,X).

The first constraint says that if list *Y* is the result of reversing list *X* then any two consecutive elements of *X* should be consecutive but in reverse order in *Y*. The first constraint in ic2 says that the reversed list has the same length as the original one.

In the experiments conducted, the system SKILit + MONIC obtains better results in terms of success rate than SKILit with negative examples (Section 6.2). For the integrity constraint in ic1, the synthesis of *rv/2* reaches 100% success rate with only 10 randomly chosen positive examples. The results with the predicate *append/3* also beat the results obtained with negative examples, mainly for the sets of integrity constraints ic1 and ic2 (see Figure 7.1).

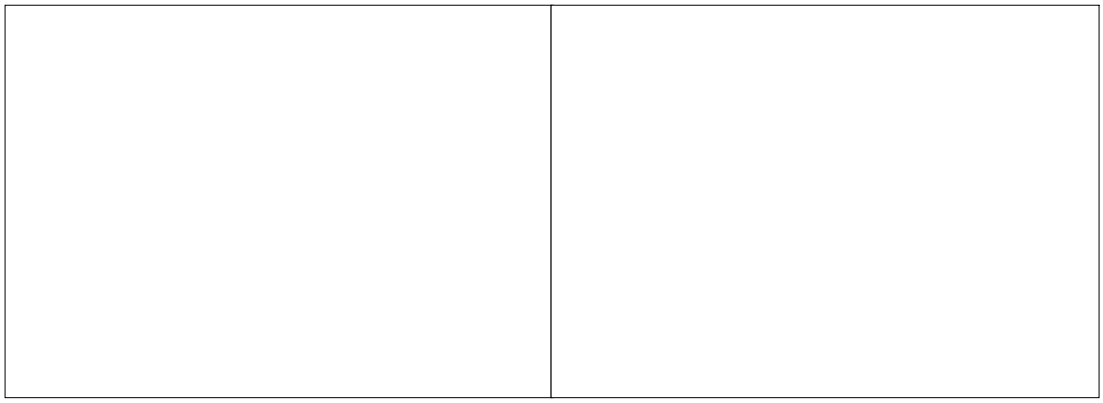

Figure 7.1: SKILit + MONIC: obtained success rate.

In terms of percentage of test-perfect programs synthesized, the results were also better than the ones obtained with negative examples for most of the integrity constraint sets chosen. This means that when the user has integrity constraints available, the chance of synthesizing the intended program increases (see Figure 7.2)

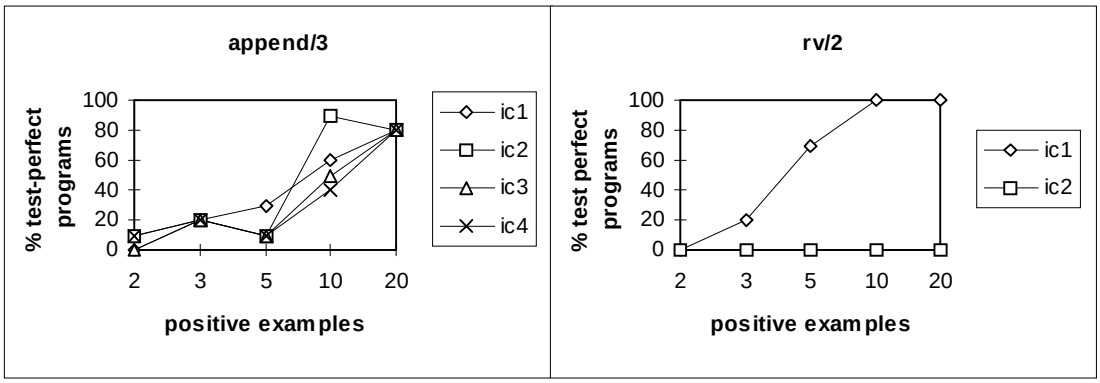

Figure 7.2: SKILit + MONIC: percentages of test-perfect program.

Similarly as with positive and negative examples, the choice of adequate integrity constraint is very important. The set of constraints ic2 for the predicate *rv/2* did not obtain good results, because those constraints do not cover many important negative examples, such as *rv([1,2],[1,2])* (they only say that the two arguments of *rv(X,Y)* should have the same number of elements and that every element of the input list is an element of the output list and vice versa).

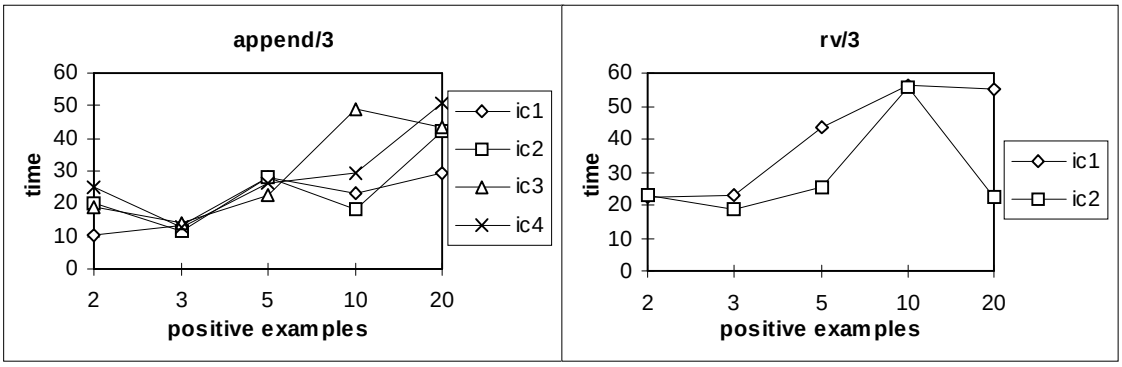

Figure 7.3: SKILit + MONIC: CPU time spent (seconds).

As we can see in Figure 7.3 the cost of using integrity constraints, in terms of time spent to synthesize a definition, is comparable or better to the exclusive use of negative examples (Section 6.2).

7.5.2 union/3

For the predicate *union/3* we used two sets of integrity constraints.

ic1: *union(A,B,C),member(X,A)→member(X,C).*
union(A,B,C),member(X,B)→member(X,C).
union([X|B],C,D),member(X,C),D=[X|E]→ C=[X|F].

ic2: *union(A,B,C),member(X,A) →member(X,C).*
union(A,B,C),member(X,B) → member(X,C).
union(A,B,C),member(X,A),member(X,B),append(A,B,C) → false.

In ic1 the first constraint says that, for every fact *union(A,B,C)*, every element of the list *A* must be in *C*. The second constraint says that every element of *B* must be in *C*. The third constraint applies when the first element *X* of the first input list is a member of the other input list (*C*) and the first element of the output list (*D*). In this case *X* must also be the first element of *C* (*union([1],[1,2],[1,2])* is a positive example while *union([1],[2,1],[1,2,1])* is a negative one).

In ic2, the first two constraints are the same as in ic1. The third constraint says that if *A* and *B* have a common element then the result of *union/3* must be different from the result of appending the two lists (*union([2],[1,2],[1,2])* and *append([2],[1,2],[2,1,2])*).

These experiments with *union/3* were conducted in a setting similar to the one used in Section 6.3. For each set of constraints, we ran SKILit+MONIC 50 times giving each time 30 positive examples randomly taken from U2i(+). No negative examples were given. We also used two different clause structure grammars: decomp_test_rec_comp_2, and decomp_+test_rec_comp_2 which forces synthesized clauses to have test literals.

constraint set	*CSG*	*success*	*test-terfect*	*time*
ic2	decomp_test_rec_comp_2	0.689	24	251.802
ic1	decomp_test_rec_comp_2	0.758	38	147.188
ic2	decomp_+test_rec_comp_2	0.780	44	122.487

Table 7.1: Experiments with *union/3* and integrity constraints.

The results are shown in Table 7.1. The success and percentage of test-perfect programs obtained with the integrity constraints in ic1 and ic2 were clearly higher than the results obtained with random negative examples (Table 6.2), and very similar to the ones obtained with the manually chosen negative examples. In the case of ic2 with the decomp_+test_rec_comp_2 CSG, the results obtained were clearly better than the ones with chosen negative examples. The synthesis time was comparable to the experiments in Section 6.3.

7.6 Related work

There are other procedures for integrity constraint checking. Sadri and Kowalski proposed a proof procedure that checks integrity constraint satisfaction, given a logic program and a set of constraints [105]. This procedure verifies if an update to the program violates any of the constraints. In case there is only one integrity constraint, the method of Sadri and Kowalski is equivalent to SLDNF-resolution (Section 3.2.3). Another theorem-prover, SATCHMO, was proposed by Manthey and Bry [65].

This kind of integrity checkers uses a systematic approach to search for inconsistencies. To find a violating instance of an integrity constraint, a large number of possible instantiations must be considered. The Monte Carlo strategy only considers a sample of those instantiations which drastically reduces the search effort for a violating instance of the integrity constraint.

Luc De Raedt used integrity constraints in system CLINT [21]. In his work, an integrity constraint is checked by transforming it into a query and posing it to the generated program.

7.7 Discussion

7.7.1 The number of queries

The proposed integrity checker finds a violating instance of an integrity constraint *I* by generating a limited number of queries. Each query is posed to program *P*. From the answers the integrity checker obtains facts. With these facts we try to obtain a violating instance of constraint *I*. If such an instance is found, we can be certain that the program and the constraint are inconsistent; otherwise, we cannot be sure they are consistent. MONIC is thus incomplete.

Nevertheless, the binomial distribution tells us that the probability, after *n* queries, of finding an inconsistency is $\alpha = (1\text{-}p)^n$, where *p* is the probability of a given query obtained from *I* and posed to program *P* to succeed. This distribution tells us that, after *n* successful queries, the higher the value of *n*, the more probable it is that *P* and *I* are consistent.

Intuitively, the value of *p* measures the consistency level of the integrity constraint with the program *P*. The actual value of *p* is unknown. However, we can choose a lower limit for *p* meaning that if there is an inconsistency, then at least $100 \times p\%$ of the queries give a violating instance of *I*. We call this the “integrity constraint generality assumption”.

After n non-violating queries, and given a value for p, we can be $100 \times (1-\alpha)\%$ sure that P and I are consistent.

Unfortunately, the user cannot say, a priori, whether this assumption will be verified or not, since that depends on P, which is unknown. We can, however, have an intuitive notion of the generality of a given constraint, and, therefore, prefer more 'general' constraints, such as *sort(X,Y)→sorted(Y)*, to more 'specific' ones, such as *sort([2,3],[3,2])→false*.

7.7.2 Soundness and completeness

When MONIC finds a violating instance of an integrity constraint, that means that the program does not satisfy the constraint. In that sense MONIC is correct or sound, since it does not find false inconsistencies. However, MONIC is only as correct as the proof procedure that it is using to answer the queries. To guarantee the integrity of the proof procedure a *safe computational rule* should be used [46].

As we previously saw, MONIC may not find a violating instance even if one does exist. For this reason, it is *incomplete*. However, when MONIC does not find an inconsistency we have an associated confidence level (α). Moreover, we can control this confidence level by choosing an appropriate number of queries.

7.7.3 Limitations

The inclusion of integrity constraints in specifications, independently of the constraint checking method employed, represents an additional effort for the user, since writing a set of adequate integrity constraints can be as complex as writing the program itself. However, the inclusion of integrity constraints in the specification is optional. If they are easy to express, they can be exploited by the system. Otherwise the system can use negative examples only.

The most important limitations, specific to our Monte Carlo approach, are the incompleteness of the search for a violating instance (discussed in Section 7.7.2), the need of associating distribution of probability functions to the involved types (Section 7.4.3), and the fact that the method does not cope with all integrity constraints (Section 7.4.1).

Even though the distributions associated to types can affect the behaviour of the method, we did not have to make any tuning of this distribution in order to achieve the obtained results. For this reason, and in practical terms, this limitation does not seem to be significant.

The restrictive integrity constraints handled by MONIC seem the most adequate for the representation of negative information in a context of synthesis from incomplete specifications. For this reason, we did not consider other sorts of constraints. Such an extension could be considered in the future.

8. Conclusion

8.1 Summary

In this thesis we present a methodology for the automatic construction of logic programs. This methodology is the basis of system SKILit. To obtain a program *P* the user supplies certain information to the system describing some aspects of that program. The system assumes that the information given is *incomplete*. From this information, which we call *specification*, SKILit builds a program *P'* which satisfies the specification. In case *P'* does not satisfy the user, he/she can supply more data to the specification and rerun SKILit. SKILit is *inductive*, for it starts from a given incomplete specification *S* and synthesizes a program *P* which may have logical consequence not described in *S*. In short, SKILit is a system for the inductive synthesis of logic programs from incomplete specifications (Chapters 4 and 5). A specification given to SKILit contains different sorts of data. The core of the specification is composed of the *positive examples* and the *negative examples*.

The *iterative induction* strategy used by SKILit (Chapter 5) helped find an answer for an important limitation of many ILP systems, representing state-of-the-art such as GOLEM

[82], FOIL [96] and Progol [80]. Unlike these systems, SKILit is capable of synthesizing recursive programs from sparse sets of positive examples. To be successful in the synthesis task, the user of SKILit does not have to supply a list of examples as complete as possible, nor to guess the examples according to possible resolution paths of the target program. Assuming that the user gives complete sets of examples or examples computationally related to each other would be against the spirit of programming by examples. Iterative induction represents an advance in the state of the art of recursive program synthesis.

While being able to cope with incomplete sets of examples, SKILit has a quite wide *class of synthesizable programs*. Other systems previous to SKILit, like LOPSTER [60], FORCE2 [12] and CRUSTACEAN [1], are also capable of synthesizing recursive definitions from few examples. However, all of them synthesize programs within a very restricted and well defined class. The strategies used by these systems, however interesting, force them to strongly restrict the language bias. Programs such as *quicksort/2*, which has two recursive literals in a clause, and *union/3* that has two recursive clauses were synthesized by SKILit, but are not synthesizable by the systems mentioned above.

The class of programs synthesizable by SKILit for a given synthesis task can be defined by a *clause structure grammar* (CSG). Although it is not strictly necessary, a CSG enables a more efficient construction of the intended program. A CSG serves to transmit to SKILit a certain programming strategy, such as “divide-and-conquer”, “generation-and-test”, etc. Since each strategy serves a vast set of programs, the CSG are highly reusable.

The *background knowledge* (BK) has an important role in the synthesis process. The predicates defined in the BK determine the vocabulary which SKILit will use in the construction of clauses. Therefore, an adequate background knowledge can turn a hard synthesis task into an easy one. On the other hand, an unsuccessful attempt may be due to insufficient background knowledge (Section 4.8).

The construction of a clause is made by searching for a *relational link* between the input arguments and the output arguments of a positive example. This strategy employed by SKILit enables the background knowledge to constrain the construction of a clause since only the logical consequences of the BK, the positive examples and of the clauses meanwhile synthesized by SKILit can be part of the link.

The clause construction strategy used by SKILit allows the BK to be defined *intensionally* (Section 4.4) as any Prolog program. Intensional BK makes the construction and maintenance of auxiliary predicates much easier.

The user of SKILit can specify the intended program entirely by examples. However, other means of specification are available. A *sketch* (Section 4.5.1), for example, provides abstracted information about how a particular example is processed. SKILit explores the given sketches by consolidating and transforming them into operational clauses (Chapter 5). Sketches and examples are handled in a uniform way. The sketch refinement operator employed by SKILit is shown to be complete under appropriate assumptions.

A system demanding an excessive number of positive examples is as inadequate for program synthesis, as a system which demands an excessive number of negative examples. For that reason, we have extended SKILit with the integrity checker MONIC which is capable of handling *integrity constraints*. Due to its expressiveness, an integrity constraint can replace a large number of ground negative examples.

The *Monte Carlo strategy* which was designed to handle constraints seems quite efficient, which makes it appropriate in an inductive synthesis context. Other existing approaches to integrity checking are heavier and could be impractical in this context.

8.2 Open problems

SKILit did not yet result in a useful support tool for a logic programming environment. However, we took some steps in the direction of making the inductive approach to program synthesis feasible. Many important problems still have to be solved. The solutions we propose could, of course, be improved. In this Section we discuss some unresolved problems and the research lines they can lead to.

8.2.1 The selection of auxiliary predicates

An inductive synthesis system should have a quite rich background knowledge, so that it could be used in a wide variety of problems. In the experiments usually carried out with ILP systems, the BK defines the exact set of auxiliary predicates needed so that the synthesis task would succeed.

Even though the strategy of following a relational link enables SKILit to filter many irrelevant auxiliary predicates, the system's performance degrades when the number of admissible auxiliary predicates is very large (Section 3.4.5). The solution adopted here involves the user who indicates to the system which are the admissible predicates to be considered. It is obvious that this is not entirely satisfactory, as it places a burden on the user.

SKILit also fails if the BK predicates are insufficient, because it cannot invent predicates. Predicate invention is a difficult task in itself [114]. Nevertheless, it would be worthwhile to extend SKILit, because a practical system should try to fill in the gaps within BK.

8.2.2 Interaction

When the specification supplied by the user is insufficient for SKILit to build an adequate program, the solution is to examine SKILit’s results and change the specification accordingly.

The synthesis system should guide the user in the construction and refinement of the specification. Interactive systems typically suggest the user which examples to supply during the synthesis process itself, freeing the user from the difficult task of guessing which are good and bad examples. Unfortunately, interactive systems tend to ask too many questions during the synthesis session, disturbing the user. For that reason, we have adopted a non interactive solution.

Nevertheless, it would be important for a system such as SKILit to have some interactive tools for evaluating the produced programs, as well as for the debugging of specifications. This sort of post-synthesis interaction would be less disturbing for the user because it would be restricted to the situations in which the final induced program is either incorrect or incomplete.

8.2.3 Many examples

SKILit is guided towards the synthesis from small sets of examples. Each clause is constructed to cover one positive example, and the remaining positive examples are not directly taken into account. SKILit explores the internal structure of the example itself, but ignores the patterns which may be common to different examples. For that reason SKILit may have difficulties in adequately coping with large quantities of data.

8.3 Evaluation of the approach

The experiments with SKILit, carried out in Chapter 6, showed that the system is capable of synthesizing list handling predicates from sets of naturally chosen positive examples. The so called natural choice of examples was simulated by randomly extracting examples from a given universe with a pre-defined distribution.

8.4 Main contributions to the state-of-the-art

We presented the notion of algorithm sketch as a formalism for partially describing computations. We defined the notions of skecth refinement and sketch consolidation and presented a refinement operator that finds all the operational consolidations of one sketch. This refinement operator is the basis of the inductive engine of our logic program synthesis methodology.

The iterative induction developed and employed in SKILit allows the synthesis of recursive definitions from small and sparse sets of examples. This is an important result, considering that SKILit does not impose very strong constraints upon the class of synthesizable programs. In particular, SKILit does not assume that the target program is recursive (unless the clausal grammar states otherwise). Recursive solutions are preferred to non-recursive ones only if the latter involve shorter clauses. This is an important feature that we do not find, for instance, in CRUSTACEAN, TIM or SYNAPSE. SKILit can also cope with very small sets of examples, because the methodology does not depend on some coverage-based or information gain oriented heuristic (Section 3.4.5).

The class of synthesizable programs can be defined through a clause structure grammar. This grammar allows the representation of generic programming knowledge, and restricts the search space. This results in more efficient induction. Nevertheless, the methodology is able to work without any grammar.

The clause structure grammars employed in SKILit allow the definition of admissible predicate sequences in the body of induced clauses. The arguments of those predicates are handled by relational linking. The advantage is that clause structure grammars are easy to define and maintain.

Finally, the Monte Carlo strategy adapted to the integrity checking allows the specification to include integrity constraints. These can be much more expressive than traditional negative examples, enabling the user to write more concise specifications. The

main advantage of our MONIC integrity checker is its efficiency. Using SKILit+MONIC, induction from a specification with positive examples and constraints is not significantly more time consuming than using plain SKILit to induce from positive and negative examples.

8.5 The Future

In the near future, SKILit could benefit from a more efficient re-implementation. It would also be advantageous to have other search strategies besides the one used, to enable coping with large numbers of examples, using large background knowledge programs, and synthesizing more complex clauses. It would be important that the system could work with little negative information (negative examples or integrity constraints). The methodology could also better exploit the information contained in algorithm sketches.

The ideal future for a system like SKILit would be its integration into a program development environment where it could be a tool among others. A graphical interface that minimised the user's efforts in building an incomplete specification and allowing the description of algorithm sketches would be crucial. It would also be important to have the possibility of communicating with other development tools, from simple text editors to static and dynamic program analysis modules.

Inductive programming techniques may in the future be useful for naïve and experienced programmers. Programming by example may be an important specification paradigm for naïve programmers that need programming for interfacing with complex applications of daily use like word processors, spreadsheets or database management systems. Inductive techniques will keep unwilling programmers away from code as much as possible. Experienced programmers may benefit from tools based on programming by example embedded in enhanced text editors. Incomplete specifications may be attached to the code and be used for program synthesis and verification.

In its core, automatic program synthesis will have to be more knowledge based. Different aspects of programming knowledge will be encoded as input for the synthesis systems. These different types of knowledge will be, on the one hand, separately represented and on the other will have to be combined to obtain, for each task, an integrated solution. In this thesis we followed this direction focussing on two types of programming knowledge: specific knowledge (sketches) and generic knowledge (structure grammars).

Automatic programming systems will have to have knowledge about auxiliary programs and about synthesized programs. Having a list of background predicates is not enough, especially if it is a very long list. An automatic programming system should, for example, easily recognize positive examples of a program it already knows.

To sum up, automatic programming provides very relevant research challenges. Their relevance derives from the importance of programming itself. Any tool or methodology that makes programming easier means that more powerful applications may appear. Either because the development of the application was made simpler or because its interface made it easier to use. More powerful and easier to use applications mean that computer-aided tasks (programming included) will be accomplished more efficiently, leaving whoever has to fulfil those tasks with more precious spare time.

References

[1] Aha, D. W., Lapointe, S., Ling, C. X., Matwin S (1994): "Inverting Implication with Small Training Sets". Proceedings of the European Conference on Machine Learning, ECML-94. Ed. F. Bergadano, L. De Raedt. Springer Verlag.

[2] Azevedo, R., Costa, V.S., Damas, L., Reis, R. (1990): "YAP Reference Manual". Centro de Informática da Universidade do Porto.

[3] Banerji, R. B. (1964): "A Language for the Description of Concepts". *General Systems*, 9, pp. 135-141.

[4] Baroglio, C., Giordana, A., Saitta, L. (1992): "Learning Mutually Dependent Relations". *Journal of Intelligent Information Systems*, 1, pp. 159-176. Kluwer Academic Publishers, Boston.

[5] Bergadano, F., (1993): "Towards an Inductive Logic Programming Language". *Deliverable no. TO1 of ILP project*.

[6] Bergadano, F., Gunetti, D. (1993): "The Difficulties of Learning Logic Programs with Cut". *Journal of Artificial Intelligence Research* 1, pp. 91-107.

[7] Biermann, A. W., (1978): "The inference of regular LISP Programs from Examples". *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-8, No. 8, August 1978.

[8] Biermann, A. W., (1990): "Automatic Programming". *Encyclopedia of Artificial Intelligence*. Ed. Stuart C. Shapiro. Wiley Interscience.

[9] Blum, L. and Blum, M. (1975): "Toward a Mathematical Theory of Inductive Inference". *Information and Control*, 28, pp. 125-155.

[10] Bratko, I. (1986): *Prolog Programming for Artificial Intelligence*. Addison-Wesley.

[11] Bratko, I., Muggleton, S., Varšek, A. (1992): "Learning Qualitative Models of Dynamic Systems". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press.

[12] Brazdil, P., Jorge, A. (1992): "Modular Approach to ILP: Learning from interaction between Modules". *Logical Approaches to Machine Learning, Workshop notes*. ECAI 92.

[13] Brazdil, P. (1981): *A Model for Error Detection and Correction*. PhD Thesis. University of Edinburgh.

[14] Brazdil, P., Jorge, A. (1994): "Learning by Refining Algorithm Sketches". *Proceedings of ECAI-94*. Ed. T. Cohn. Wiley.

[15] Brazdil, P., Jorge, A. (1997): "Induction with Subtheory Selection". *ECML 97 - Poster Papers*. Ed. M. van Someren, G. Widmer. Laboratory of Intelligent Systems, Faculty of Informatics and Statistics, University of Economics, Prague.

[16] Buntine, W. (1988): "Generalized Subsumption and its Application to Induction and Redundancy". *Artificial Intelligence* 36, pp 149-176, Elsevier Sience Publishers B.V. (North Holland).

[17] Calejo, M. (1991): *A Framework for Declarative Prolog Debugging*. PhD Thesis. Universidade Nova de Lisboa.

[12] Cohen, W. W. (1993): "Pac_learning a restricted class of recursive logic programs". *Proceedings of the third International Workshop on Inductive Logic Programming* (pp. 73-86). Bled, Slovenia. J. Stefan Institute.

[18] Cohen, W. W. (1993): "Rapid prototyping of ILP systems using explicit bias". *Proceedings of 1993 IJCAI Workshop on ILP*.

[19] Cypher, A. (Ed.) (1993): *Watch What I Do: Programming by Demonstration*. MIT Press.

[20] De Raedt, L. (1991): *Interactive Concept Learning*. PhD thesis. Katholieke Universiteit Leuven.

[21] De Raedt, L. (1991): *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press.

[22] De Raedt, L., Bruynooghe, M. (1993): "A theory of Clausal Discovery". *Proceedings of IJCAI-93*. Chamberry, France.

[23] De Raedt, L., Idestam-Almquist, P, Sablon, G (1997): "θ-subsumption for Structural Matching". *Proceedings of ECML-97*. Prague. M. van Someren, G. Widmer (Ed.). Springer.

[24] De Raedt, L. Lavrac, N. (1995): "Multiple Predicate Learning in two Inductive Logic Programming settings". *Journal of Pure and Applied Logic*, 4(2):227-254.

[25] De Raedt, L., Lavrac, N., Dzeroski, S. (1993): "Multiple Predicate Learning". *Proceedings of IJCAI-93*. Chamberry. France. R. Bajcsy (Ed.). Morgan Kaufmann.

[26] Deville, Y. (1990): *Logic Programming, Systematic Program Development*. Addison-Wesley Publishing Company.

[27] Deville, Y., Lau, K.,(1994): "Logic Program Synthesis". *The Journal of Logic Programming, special issue Ten Years of Logic Programming,* volumes 19,20, May/July 1994.

[28] Diller, A. (1991): *Z, an introduction to formal methods*. Wiley.

[29] Dolšak, B. and Muggleton, S. (1992): "The Application of Inductive Logic Programming to Finite Element Mesh Design". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press.

[30] Dromey, G. (1989): *Program Derivation, the development of programs from specification*. Addison-Wesley.

[31] Ducassé, M. (1994): "Logic Programming Environments: Dynamic Program Analysis and Debugging". *The Journal of Logic Programming, special issue Ten Years of Logic Programming,* volumes 19,20, May/July 1994.

[32] Esposito, F., Malerba, G., Semeraro, G. and Pazzani, M. (1993): "Document understanding: a machine learning approach". *Real-World Applications of Machine Learning, Workshop notes*. Ed. Y. Kodratoff, P. Langley. ECML-93, Vienna.

[33] Feng, C. (1992): "Inducing Temporal Fault Diagnostic Rules from a Qualitative Model". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press.

[34] Feng, C. and Muggleton, S. (1992): "Towards Inductive Generalization in Higher Order Logic". *Proceedings of the Ninth International Workshop ML92*. Ed. Derek Sleeman and P. Edwards. Morgan Kaufmann.

[35] Fisher, A., (1988): *CASE: Using Software Development tools*. Wiley.
[36] Flach, P. (1995): *Conjectures: an inquiry concerning the logic of induction*. PhD Thesis. ITK dissertation series - 1.
[37] Flener, P. (1995): *Logic Program Synthesis From Incomplete Information*. Kluwer Academic Publishers.
[38] Flener, P., Deville, Y. (1992): "Logic Program Synthesis from Incomplete Specifications". Research Report RR 92-22. Université Catholique de Louvain, Unite d'Informatique.
[39] Flener, P., Popelínský, L., (1994): "On the use of Inductive Reasoning in Program Synthesis: Prejudice and Prospects". *Joint Proc. of LOPSTR'94 and META'94,* LNCS, Springer-Verlag.
[40] Giordana, A., Saitta, L., Baroglio, C. (1993): "Learning Simple Recursive Theories". *Proceedings of the 7th International Symposium, ISMIS'93*. Lecture Notes in Artificial Intelligence. Springer-Verlag.
[41] Grobelnik, M. (1992): "MARKUS: An Optimal Model Inference System". *Proceedings on ECAI-92 Workshop on Logical Approaches to Machine Learning*. Rouveirol, C. (Ed.).
[42] Heidorn, G. E. (1975): "Automatic Programming Through Natural Language Dialogue: A Survey". [99].
[43] Helft, N. (1987): "Inductive Generalization: a Logical Framework". *Proceedings of EWSL 87*. Ed. I. Bratko, N. Lavrac. Sigma Press.
[44] Helft, N. (1989): "Induction as nonmonotonic inference". *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pp 149-156. Morgan-Kaufmann.
[45] Hoare, C.A.R (1986): "Mathematics of Programming". BYTE 11(8).
[46] Hogger, C. J. (1990): *Essentials of Logic Programming*. Graduate texts in computer science series. Oxford University Press.
[47] Idestam-Almquist, P. (1993): "Generalization under implication by recursive anti-unification". *Proceedings of ILP-93*. Technical Report. Jozef Stefan Institute.
[48] Idestam-Almquist, P. (1993): *Generalization of Clauses*. PhD thesis. Report Series No. 93-025. Stockholm University, Royal Institute of Technology, Department of Computer and Systems Sciences.
[49] Idestam-Almquist, P. (1995): "Efficient Induction of Recursive Definitions by Structural Analysis of Saturations". *Proceedings of the Fifth International Workshop on Inductive Logic Programming*. Ed. L. De Raedt. Scientific Report. Departement of Computer Science, K.U. Leuven.
[50] Jaquet, J.-M. (Ed.) (1993): *Constructing Logic Programs*. Wiley Professional Computing. Wiley.
[51] Jazza, A. (1995): "Toward Better Software Automation". *Software Engineering Notes*, vol. 20 no. 1. ACM SIGSOFT.
[52] Joch, A. (1995): "How Software doesn't work". BYTE, 20(12).
[53] Jorge, A. and Brazdil, P. (1996): "Architecture for Iterative Learning of Recursive Definitions". *Advances in Inductive Logic Programming*. Ed. Luc De Raedt. IOS Press.

[54] Jorge, A. and Brazdil, P. (1996): "Integrity Constraints in ILP using a Monte Carlo approach". *Inductive Logic Programming, 6th International Workshop, ILP-96*. Ed. Stephen Muggleton. LNAI 1314. Springer..

[55] Jüllig, R. K. (1993): "Applying Formal Software Synthesis". *IEEE Software*. Vol. 10, No. 3, pp. 11-22.

[56] Kietz, J. and Wrobel, S. (1992): "Controlling the Complexity of Learning in Logic". *Inductive Logic Programming*. Ed. Stephen Muggleton. Academic Press Limited.

[57] Kijsirikul, B., Numao M. and Shimura, M. (1992): "Discrimination-Based Constructive Induction of Logic Programs". *Proceedings of AAAI-92*. Morgan-Kaufmann.

[58] Klingspor, V. (1994): "GRDT: Enhancing Model-Based Learning for Its Application in Robot Navigation". *Proceedings of the Fourth International Workshop on Inductive Logic Programming (ILP-94)*. GMD-Studien Nr. 237. GMD, Alemanha.

[59] Korf, R. E., (1990): "Search". *Encyclopedia of Artificial Intelligence*. Ed. Stuart C. Shapiro. Wiley Interscience.

[60] Lapointe, S., Matwin, S., (1992): "Sub-unification: A tool for efficient induction of recursive programs". *Proceedings of the Ninth International Conference on Machine Learning* (pp. 273-281). Aberdeeen, Scotland. Morgan Kaufmann.

[61] Lavrac, N., Dzeroski, S. (1992): "Inductive Learning of Relations from Noisy Examples". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press Limited.

[62] Lavrac, N., Dzeroski, S. (1994): *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

[63] Ling, C. X., (1991): "Inductive Learning From Good Examples". *12th International Joint Conference on Artificial Intelligence*. Ed. J. Mylopoulos, R. Reiter. Morgan Kaufmann.

[64] Lloyd, J. W., (1987) *Foundations of Logic Programming* (second, extended edition). Springer-Verlag.

[65] Manthey, R., Bry, F. (1988): "SATCHMO: a theorem prover implemented in Prolog". *Proceedings of CADE 88 (9th Conference on Automated Deduction)*. Springer-Verlag.

[66] Michalski, R. S. (1983): "A Theory and Methodology of Inductive Learning". [69].

[67] Michalski, R. S. (1990): "Learning, Machine". *Encyclopedia of Artificial Intelligence*. Ed. S. Shapiro. Wiley Inter-Science.

[68] Michalski, R. S. (1994): "Inferential Theory of Learning: Developing Foundations for Multistrategy Learning". *Machine Learning, A Multistrategy Approach, Volume IV*. Ed. Ryszard Michalski, Gheorghe Tecuci. Morgan Kaufmann.

[69] Michalski, R. S., Carbonell, J. and Mitchell, T. (1983) *Machine Learning: An Artificial Intelligence Approach*. Tioga Publishing Company.

[70] Michalski, R. S., Larson, J. B. (1978): "Selection of most representative training examples and incremental generation of VL1 hypotheses: The underlying methodology and description of programs ESEL and AQ11". Technical report 867. Computer Science Department, University of Illinois, Urbana-Champaign.

[71] Minker, J. (Ed.) (1988): *Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers.

[72] Mitchell, T. (1982): "Generalization as Search". *Artificial Intelligence*, 18, pp. 203-226.

[73] Mitchell, T. (1990): "The need for biases in learning generalizations". *Readings in Machine Learning.* Ed. J. Shavlik and T. Dietterich. Morgan Kaufmann.

[74] Mofizur, C. R. and Numao, M. (1995): "Top-down Induction of Recursive Programs from Small Number of Sparse Examples". *Proceedings of the Fifth International Workshop on Inductive Logic Programming*. Ed. L. De Raedt. Scientific Report. Departement of Computer Science, K.U. Leuven.

[75] Morik, K., Potamias, G. and Moustakis, V. (1993): "Knowledgeable Learning Using MOBAL - A Case Study in A Medical Domain". *Real-World Applications of Machine Learning*, Workshop notes. Ed. Y. Kodratoff, P. Langley. ECML-93, Viena.

[76] Morik, K., Wrobel, S., Kietz, J. and Emde, W. (1993): *Knowledge Acquisition and Machine Learning: Theory Methods and Applications.* Academic Press.

[77] Muggleton, S. (1994): "Inverting Implication". Preliminary version.

[78] Muggleton, S. (1992): "Inductive Logic Programming". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press. Also in *Proceedings of the First International Conference on Algorithmic Learning Theory*, Ohmsha, Tokyo, 1990.

[79] Muggleton, S. (1993): "Inductive Logic Programming: derivations, successes and shortcomings". *Proceedings of ECML-93*. Ed. P. Brazdil. Springer-Verlag.

[80] Muggleton, S. (1995): "Inverse Entailment and Progol". *New Generation Computing Journal*, vol. 13, May 1995.

[81] Muggleton, S. (1995): "Stochastic Logic Programs: extended abstract". *Proceedings of ILP-95*. Ed. Luc De Raedt. Scientific Report. Katholiek Universiteit Leuven.

[82] Muggleton, S., Feng, C. (1990): "Efficient Induction of Logic Programs". *Proceedings of the 1st Conference on Algorithmic Learning Theory*, Ohmsha, Tokyo.

[83] Muggletton, S., De Raedt, L., (1994): "Inductive Logic Programming". *The Journal of Logic Programming, special issue Ten Years of Logic Programming*. Vol. 19,20, May/July 1994.

[84] Muggleton, S., Mizoguchi, F. and Furukawa, K. (1995): Preface of the Special Issue on Inductive Logic Programming, *New Generation Computing*. Vol. 13, Nos. 3,4. Springer-Verlag.

[85] Muggleton, S., King, R., Sternberg, M. (1992): "Protein secondary structure prediction using logic". *Proceedings of the 2nd International Workshop on ILP*. Muggleton, S. (Ed.). Report ICOT-TM 1182, pp. 228-259.

[86] Muggleton, S., King, R., Sternberg, M. (1992): "Protein secondary structure prediction using logic". *Protein Engineering*. 7:647-657.

[87] Nienhuys-Cheng, S-H., de Wolf, R. (1995): "Least Generalizations and Greatest Specializations of Sets of Clauses". *Journal of Artificial Intelligence Research*. Volume 4, pp 341-363.

[88] O'Keefe, R. (1990): *The Craft of Prolog*. MIT Press.
[89] Olsson, R. (1995): "Inductive Functional Programming Using Incremental Program Transformation". *Artificial Intelligence* 74, pp 55-81. Elsevier.
[90] Paakki, J., Gyimóthy, T. and Horváth, T. (1994): "Effective Algorithm Debugging for Inductive Logic Programming". *Proceedings of the Fourth International Workshop on Inductive Logic Programming (ILP-94)*. GMD-Studien Nr. 237. GMD, Alemanha.
[91] Pereira, L. M. and Calejo, M. (1989): "Algorithmic Debugging of Prolog Side-Effects". *Proceedings of EPIA 89*, ed. by J. P. Martins and E. Morgado, Lecture Notes in Artificial Intelligence, Springer-Verlag.
[92] Plotkin, G. (1969): "A note on inductive generalization". *Machine Intelligence 5*. Ed. B. Meltzer, D. Michie. Edinburgh University Press.
[93] Plotkin, G. (1971): "A further note on inductive generalization" . *Machine Intelligence 6*. Ed. B. Meltzer, D. Michie. Edinburgh University Press.
[94] Popelínský, L., Flener P, Stepánková O, (1994) "ILP and Automatic Programming: Towards Three Approaches". *Proceedings of the 4th International Workshop on Inductive Logic Programming*. Volume 237, GMD-Studien.
[95] Popelinský, L., Stpánková, O. (1995): "WiM: A study on top-down ILP program". *Proceedings of AIT'95 Workshop*. Brno. ISBN 80-214-0673-9.
[96] Quinlan, J. R. (1990): "Learning Logical Definitions from Relations". *Machine Learning 5(3)*, pp.239-266.
[97] Quinlan, J. R. and Cameron-Jones, R. M. (1993): "FOIL: A Midterm Report". *Proceedings of the European Conference on Machine Learning ECML-93*. Ed. P. Brazdil. Springer-Verlag.
[98] Quinlan, J. R. and Cameron-Jones, R. M. (1995): "Induction of Logic Programs: FOIL and related systems". *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4).
[99] Rich, C. and Waters, R. (Eds.) (1986): *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann.
[100] Richards, B., Mooney, R. (1995): "Refinement of First-Order Horn Clause Domain Theories". *Machine Learning*,Vol.19, No.2. Kluwer Academic Publishers.
[101] Richards, B., Mooney, R. (1992): "Learning relations by pathfinding". *Proceedings of the Tenth National Conference on Artificial Intelligence*. MIT Press.
[102] Rouveirol, C. (1992): "Extensions of Inversion of Resolution Applied to Theory Completion". *Proceedings of the 1st International Workshop on Inductive Logic Programming*. Technical Report. LIACC, Universidade do Porto.
[103] Rubinstein, R. (1981): *Simulation and the Monte Carlo Method*. John Wiley & Sons.
[104] Russel, S., Subramanian, G. (1990): "Mutual Constraints on Representation and Inference". *Machine Learning, Meta-Reasoning and Logics*. Ed. P. Brazdil, K. Konolige. Kluwer Academic Press.
[105] Sadri, F., Kowalski, R. (1988): "A Theorem Proving Approach to Database Integrity". *Deductive Databases and Logic Programming*. Ed. Jack Minker. Morgan Kaufmann Publishers.

[106] Sammut, C. (1993): "The Origins of Inductive Logic Programming: A Prehistoric Tale". *Proceedings of ILP-93*. Technical report. Jozef Stefan Institute.

[107] Sammut, C., Banerji, R. (1986): "Learning Concepts by asking Questions". *Machine Learning: An Artificial Intelligence Approach*, Vol. 2. Ed. R. Michalski, J. Carbonell, T. Mitchel. Morgan Kaufmann.

[108] Sernadas, A., Sernadas, C. and Costa, J.F. (1995): "Object-Specification Logic". *Journal of Logic and Computation* 5(5), pp. 603-630.

[109] Shapiro, E. Y., (1982) *Algorithmic Program Debugging*. MIT Press.

[110] Silverstein, G. and Pazzani, M. (1989): "Relational Clichés: constraining constructive induction during relational learning". *Proceedings of the Sixth International Workshop on Machine Learning*, Evanston, Illinois. Kaufmann.

[111] Smith, D. R., (1990): "KIDS: A Semiautomatic Program Development System". *IEEE Trans. on Software Engineering*, Vol. 16 No. 9.

[112] Solomonoff, R. J. (1964): "A formal Theory of Inductive Inference, Part I". *Information and Control*, 7, pp. 1-22.

[113] Sommerville, I., (1989): *Software Engineering* (third edition). Addison-Wesley Publishers Ltd.

[114] Stahl, I. (1993): "Predicate Invention in ILP - an Overview". *Proceedings of ECML-93*. Ed. P. Brazdil. Springer-Verlag.

[115] Stahl, I. (1995): "The Appropriateness of Predicate Invention as Bias Shift Operation in ILP". *Machine Learning*, 20. pp.95-117.

[116] Sterling, L., Shapiro, E. Y., (1986) *The Art of Prolog: Advanced Programming Techniques*. MIT Press.

[117] Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., Underwood, I., (1994): "Deductive composition of Astronomical Software from Subroutine Libraries". *Proceedings of the Twelfth International Conference on Automated Deduction*. Nancy, France. A. Bundy (Ed.). LNAI 814. Springer Verlag.

[118] Summers, P. (1977): "A Methodology for LISP Program Construction from Examples". *Journal of the Association for Computing Machinery*, Vol. 24, No. 1.

[119] Tausend, B. (1994): "Representing Biases for Inductive Logic Programming". *Proceedings of ECML-94*. Ed. F. Bergadano, L. De Raedt. Springer-Verlag.

[120] Tausend, B. (1994): "Biases and Their Effects in Inductive Logic Programming". *Proceedings of ECML-94*. Ed. F. Bergadano, L. De Raedt. Springer-Verlag.

[121] Ullman, J. (1989): *Principles of Databases and Knowledge Base Systems*. Volumes I and II. Computer Science Press.

[122] Vere, S. (1977): "Induction of Relational Productions in the Presence of background Knowledge". *Proceedings of the Fifth International Joint Conference in Artificial Intelligence*.

[123] Wirth, R. and O'Rorke (1992): "Constraints for predicate invention". *Inductive Logic Programming*. Ed. S. Muggleton. Academic Press.

[124] Wrobel, S. (1994): *Concept Formation and Knowledge Revision*. Kluwer Academic Publishers.

[125] Zelle, J. M., Mooney, R. J., Konvisser, J. B., (1994): "Combining Top-down and Bottom-up Techniques in Inductive Logic Programming". *Proceedings of the*

Eleventh International Conference on Machine Learning ML-94, Morgan-Kaufmann.

Annex

Appendix A

Predicates in background knowledge *list*:

*mode(**const**(+,+,-)).*
type(const(list,int,list)).

const(A,B,C)←A=[B|C].

*mode(**dest**(+,-,-)).*
type(dest(list,int,list)).

dest(A,B,C)←A=[B|C].

*mode(**null**(+)).*
type(null(list)).

null([]).

*mode(**addlast**(+,+,-)).*
type(addlast(list,int,list)).

addlast([],X,[X]).
addlast([A|B],X,[A|C])←
addlast(B,X,C).

*mode(**appendb**(+,+,-)).*
type(appendb(list,list,list)).

appendb([],A,A).
appendb([A|B],C,[A|D])←
appendb(B,C,D).

% predicate *append/3* has a definiton equivalent to *appendb/3*.

*mode(**delete**(+,+,-)).*
type(delete(elem,list,list)).

```
delete(A,[A|B],B).
delete(A,[B|C],[B|D])←
        delete(A,C,D).

mode(last_of(-,+)).
type(last_of(elem,list)).

last_of(A,[A]).
last_of(A,[C|D])←last_of(A,D).

mode(memberb(-,+)).
type(memberb(elem,list)).

memberb(A,[A|B]).
memberb(A,[B|C])←
        memberb(A,C).
```

% predicate *member/2* has a definition equivalent to *memberb/2*.

```
mode(notmember(+,+)).
type(notmember(elem,list)).

notmember(A,B)←memberb(A,B).

mode(partb(+,+,-,-)).
type(part(int,list,list,list)).
```

% predicate *partition/4* has a definition equiavalent to *partb/4*.

```
partb(A,[B|C],[B|D],E)←
        A>B,partb(A,C,D,E).
partb(A,[B|C],D,[B|E])←
        A=<B,partb(A,C,D,E).
partb(A,[],[],[]).

mode(rv(+,-)).
type(rv(list,list)).

rv( [], []).
rv( [C|D], B) ←
   rv(D,E),
   addlast(E,C,B).
```

```
mode(singleton(+)).
type(singleton(list)).

singleton([X]).

mode(sort(+,-)).
type(sort(list,list)).

sort([A|B],C)←
        part(A,B,E,F),
        sort(E,G),sort(F,H),
        append(G,[A|H],C).
sort([],[]).

mode(split(+,-,-)).
type(split(list,list,list)).

split([],[],[]).
split([A,B|D],[A|E],[B|F])←split(D,E,F).

mode(union(+,+,-)).
type(union(list,list,list)).

union([],A,A).
union([A|B],C,D)←
        member(A,C),!,
        union(B,C,D).
union([A|B],C,[A|D])←
        union(B,C,D).

mode(insertb(+,+,-)).
type(insertb(int,list,list)).

insertb(A,[],[A]).
insertb(A,[B|C],[A,B|C])←A<B.
insertb(A,[B|C],[B|D])←B<A,insertb(A,C,D).
```

Predicates in background knowledge *integer*:

```
mode(pred(+,-)).
type(pred(int,int)).
```

pred(X,Y)←Y is X-1,X>0.

*mode(**succ**(+,-)).*
type(succ(int,int)).

succ(X,Y)←Y is X+1.

*mode(**zero**(+)).*
type(zero(int)).

zero(0).

*mode(**one**(+)).*
type(one(int)).

one(1).

*mode(**plus**(+,+,-)).*
type(plus(int,int,int)).

plus(X,Y,Z)←Z is Y+X.

*mode(**multb**(+,+,-)).*
type(multb(int,int,int)).

*multb(X,Y,Z)←Z is X*Y.*

%predicate multiply/3 is equivalent to multb/3.

*mode(**even**(-)).*
type(even(peano)).

even(0).
even(s(s(X)))←even(X).

*mode(**odd**(+)).*
type(odd(peano)).

odd(s(0)).
odd(s(s(X)))← odd(X).

Other predicates:

sorted*([]).*
sorted([A]).
sorted([A,B|C])←
    A=<B,
    sorted([B|C]).

sublist*(A,B)←*
    prefix(A,B).
sublist(A,[B|C])←
    sublist(A,C).

prefix([],A).
prefix([A|B],[A|C])←
    prefix(B,C).

Appendix B

Definitions of types:

int(X)←member(X,[0,1,2,3,4,5,6,7,8,9]).

letter(X)←member(X,[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]).

list([]).
list([A|B])←int(A),list(B).

peano(0).
peano(s(X))←peano(X).

Appendix C

‘decomp_test_rec_comp_2’

body(P)-->decomp(+,2),test(,2),recurs(*,2,P),comp(*,2).*
body(P)-->test(+,2),comp(,2).*

decomp(_,N)-->lit_decomp,{N>0}.
decomp(_,N)-->lit_decomp,{N2 is N-1},decomp(+,N2).
decomp(,N)-->[].*

lit_decomp-->[dest/3];[pred/2];[partb/4].

test(_,N)-->lit_test,{N>0}.
test(_,N)-->lit_test,{N2 is N-1},test(+,N2).
test(,N)-->[].*

lit_test-->[null/1];[memberb/2];[notmemberb/2];[zero/1].

comp(_,N)-->lit_comp,{N>0}.
comp(_,N)-->lit_comp,{N2 is N-1},comp(+,N2).
comp(,N)-->[].*

lit_comp-->[appendb/3];[insertb/3];[addlast/3];[const/3].

recurs(_,N,P)-->lit_recurs(P),{N>0}.
recurs(_,N,P)-->lit_recurs(P),{N2 is N-1},rec(+,N2,P).
recurs(,N,P)-->[].*

lit_recurs(P)-->[P].

‘decomp_+test_rec_comp_2’

body(P)-->decomp(+,2),test(+,2),rec(,2,P),comp(*,2).*
body(P)-->test(+,2),comp(,2).*

Otherwise identical to decomp_test_rec_comp_2.dcg.

‘decomp_test_rec1_comp_2’

body(P)-->decomp(+,2),test,2),rec(*,1,P),comp(*,2).*
body(P)-->test(+,2),comp(,2).*

Otherwise identical to decomp_test_rec_comp_2.dcg.

List of Figures

List of Algorithms

List of Examples

List of Definitions

Index

—E—

—F—

—G—

—H—

—I—

—L—

—M—

—N—

—O—

—P—

—Q—

—R—

—S—

—*T*—

—*U*—

—*V*—