INDUCTIVE SYNTHESIS OF RECURSIVE LOGIC PROGRAMS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Serap Yılmaz
August 1997

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Ass't Prof. Pierre Flener (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Ass't Prof. Ilyas Çiçekli

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Ass't Prof. Ayşe Göker

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet Baray, Director of the Institute of Engineering and Science

ABSTRACT

INDUCTIVE SYNTHESIS OF RECURSIVE LOGIC PROGRAMS

Serap Yılmaz
M.S. in Computer Engineering and Information Science
Supervisor: Ass't Prof. Pierre Flener
August 1997

The learning of recursive logic programs (i.e. the class of logic programs where at least one clause is recursive) from incomplete information, such as input/output examples, is a challenging subfield both of ILP (Inductive Logic Programming) and of the synthesis (in general) of logic programs from formal specifications. This is an extremely important class of logic programs, as the recent work on constructive induction shows that necessarily invented predicates have recursive programs, and it even turns out that their induction is much harder than the one of non-recursive programs. We call this *inductive program synthesis*. We introduce a system called DIALOGS-II (Dialogue-based Inductive and Abductive LOgic Program Synthesizer-II) whose ancestor is DIALOGS. It is a schema-guided, interactive, and non-incremental synthesizer of recursive logic programs that takes the initiative and queries a (possibly naive) specifier for evidence in her/his conceptual language. It can be used by any learner (including itself) that detects, or merely conjectures, the necessity of invention of a new predicate. Moreover, due to its powerful codification of "recursion-theory" into program schemata and schematic constraints, it needs very little evidence and is very fast.

Keywords: program development, inductive logic programming, automatic program synthesis, schema-guided program synthesis.

ÖZET

ÖZYINELI MANTIK PROGRAMLARININ TÜMEVARIMSAL YOLLA SENTEZI

Serap Yılmaz
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yrd. Doç. Pierre Flener
Ağustos 1997

Özyineli mantık programlarının (en azından bir yantümcesi özyineli olan) tam olmayan bilgiden yola çıkılarak, mesela, girdi/çıktı örneklerinden, otomatik sentezi oldukça zor bir iştir. Ve bu iş tümevarımsal mantık programlama ile otomatik program sentezinin bir alt çalışma alanıdır. Bu tür programlar mantık programlarının çok önemli bir sınıfını oluştururlar. Yapıcı tümevarım çalışmaları göstermiştir ki özyineli programların sentezi özyineli olmayan programların sentezinden çok daha zordur. Bu çalışma alanı "tümevarımsal program sentezi" diye anılır. DIALOGS-II adıyla geliştirdiğimiz sistem (bu sistemin bir önceki versiyonu DIALOGS adlı sistemdir) taslak-yönetimli, interaktif ve artımsızdır. Sistem insiyatifi alıp kullanıcıyı kullanıcının dilinde sorgulayarak özyineli mantık programları sentezler. Sistem kendisi tarafindan özyineli olarak ya da başka bir sistem tarafindan, sistem özyineli bir programın sentezinin gerekliliğini farkettiği zaman kullanılabilir. "Özyineleme Teorisi" sistemin içinde taslaklar tarafından etkili bir şekilde kodlandığı için sistem çok az bilgiye gerek duyar ve çok hızlı çalışır.

Anahtar Sözcükler: program geliştirme, tümevarımsal mantık programlama, otomatik program sentezi, taslak yönetimli program sentezi.

ACKNOWLEDGEMENTS

I am very grateful to my supervisor, Ass't Prof. Pierre Flener, for his invaluable guidance during this study. His instruction will be the closest and most important reference in my future work. I would also like to thank Halime Büyükyıldız and Esra Erdem for sharing their work with me, my family and my dear friend Serap Fırat for their moral and motivating support.

Contents

1 Introduction 1

1.1 Terminology and Theoretical Results 2

1.1.1 Approaches and Extensions to ILP (and Inductive Synthesis) 2

1.1.2 Additional Specification Information 5

1.1.3 Syntactic Bias 6

1.1.4 Generality 8

1.1.5 Predicate Invention 10

1.1.6 Construction Modes and Admissibility 13

1.2 The Objective of the Thesis 15

2 The DIALOGS-II Technique 19

2.1 Asking For a Predicate Declaration, a Schema and a Strategy 21

2.2 Execution of the Strategy 23

2.3 Abduction of Evidence for the Open Relations of the Open Program . . 26

2.4 Induction of Clauses: The Program Closing Method 37

2.5 Evaluation of the Program Closing Method 40

2.5.1 Necessary Predicate Invention 41

2.5.2 Handling the Sparseness Problem 52

3 Comparison of DIALOGS-II with other ILP Systems 59

3.1 Comparison in Terms of the Evidence 59

3.2 Comparison in Terms of Schemata 61

3.3 Comparison of DIALOGS-II with DIALOGS 62

4 Conclusion 67

References 69

Appendix A: README file for DIALOGS-II 72

Appendix B: Sample Syntheses 74

Chapter 1

Introduction

In its most general form, the task of Inductive Logic Programming (ILP) is to infer a hypothesis H from assumed-to-be-incomplete information (or: evidence) E and background knowledge B such that $B \wedge H \models E$, where H, E, and B are sets of clauses. We say that H covers E (in B). In practice, B and H are often restricted to sets of Horn clauses (i.e. definite logic programs). Evidence E is usually divided into positive evidence E^+ and negative evidence E^-. Often, the clauses of E^+ are restricted to ground positive literals (or: atoms) and are called positive examples, whereas those of E^- are restricted to ground negative literals and are called negative examples: this yields an extensional description, whereas the hypothesis is an intensional description. In a more traditional machine learning terminology, we would say that a concept description H is to be learned from descriptions E of instances and counter-examples of concepts, whose features are represented by predicate symbols. In general thus, nothing restricts the evidence to be about a single concept, so that multiple (possibly related) concepts may have to be learned at the same time.

For instance, given the positive examples (in the left column) and negative examples (in the right column)

subset([],[])	¬subset([k],[])
subset([],[a,b])	¬subset([n,m,m],[m,n])

```
subset([d,c],[c,e,d])
subset([h,f,g],[f,i,g,h,j])
```

and given as background knowledge (among others) the logic program

```
select(X,[X|Xs],Xs) ←
select(X,[H|Ys],[H|Zs]) ← select(X,Ys,Zs)
```

a possible hypothesis is the logic program

```
subset([],Xs) ←
subset([X|Xs],Ys) ← select(X,Ys,Zs), subset(Xs,Zs)
```

though at this point we do not wonder how this could be feasible. The main issue is that we human beings can perform this kind of task, so that the question arises whether a machine can be designed to do it also. The usefulness of such a machine is undeniable as it would be a step towards a form of human/machine communication that more closely models inter-human communication, which usually features a lot of incomplete (and hence ambiguous) information, of course in the presence of background knowledge, and even noisy information. In the following two sub-sections, we will first introduce some terminology and theoretical results (Section 1.1) and next we will present our objective (Section 1.2).

1.1 Terminology and Theoretical Results

We now introduce some terminology (in Section 1.1.1 to Section 1.1.3 and in Section 1.1.6) and mention some theoretical results (in Section 1.1.4 and Section 1.1.5) concerning the induction of recursive clauses.

1.1.1 Approaches and Extensions to ILP (and Inductive Synthesis)

Whether for ILP in general or synthesis in particular, there is additional terminology due to different approaches as well as extensions to the ILP task, all of which we now discuss in a loosely connected fashion.

Often, the agent that provides the inputs to an ILP technique is called the teacher, whereas the ILP technique is called the learner and is said to perform learning. Such a machine learning terminology is misleading [17], and we shall use the more general terminology of *source*, *induction technique*, and *induction* instead.

An *intended relation* is the entire (possibly infinite) relation represented by a predicate symbol. In an ILP task, only *incomplete* information (called evidence) is available, i.e. it does not describe superset(s) of the intended relation(s). We here assume that the evidence has *correct* information, i.e. that it describes subset(s) of the intended relation(s). In this case, one also says that there is no *noise*. Often, the actually described subset(s) are finite. An extreme case of incomplete but correct information is complete and correct information, though this can often only be achieved through some (finite) axiomatization in the hypothesis language, but not in the evidence language.

We partition relations into *semantic manipulation* relations and *syntactic manipulation* relations, depending on whether the actual constants occurring in a ground tuple are relevant or not for deciding whether that tuple belongs to a relation. For instance, subset is a syntactic manipulation relation, because it treats constants like variables, whereas sort and insert would be semantic manipulation relations (see Section 1.1.5).

Induction can be viewed as *search* through a graph (or: search space) where the nodes correspond to hypotheses and the arcs correspond to hypothesis-transforming operators. As usual, the challenge is to efficiently navigate through such a search space, via intelligent control (e.g., by organizing the search space according to a partial order and using pruning techniques).

Induction may be *interactive* or *passive*, depending on whether the technique asks *questions* (or: *queries*) to some *oracle* (or: *informant*) or not. The oracle may or may not be the source. The questions may be of various kinds, such as the request for classification of invented examples as positive or negative ones.

Induction may be *incremental* or *non-incremental*, depending on whether evidence is input one-at-a-time with occasional output of (external) intermediate hypotheses, or input all-at-once with output of a unique final hypothesis (though there may be internal intermediate approximations, which are however not considered as hypotheses).

Induction may be *bottom-up* or *top-down*, depending on whether hypotheses (whether internal or external) monotonically evolve from the maximally specific one (namely the empty logic program) or from the maximally general one (namely a logic program succeeding on all possible queries).

In the output hypothesis, some predicate symbols may be recursively defined: the corresponding clauses are partitioned into *base clauses* and *recursive clauses*.

Once a hypothesis is accepted (for whatever reasons), one may want to validate it. Since there is no complete description of the intended relation(s), one can only test the hypothesis, rather than somehow mathematically verifying it. Ideally, a hypothesis covers all the given evidence. One may thus test the hypothesis by measuring its accuracy (expressed in percents) in correctly covering other evidence. The given evidence is thus also called the *training set*, whereas the additional evidence is called the *test set* and is usually in the evidence language. We here assume that the test set is also correct w.r.t. the intended relation(s).

An *identification criterion* defines the moment where an induction technique has been successful in correctly identifying the intended relation(s), whether it "knows" this or not. Sample criteria are finite identification, identification-in-the-limit, probably-approximately-correct (PAC) identification, and so on (see [21] for details). There are limiting theorems stating what hypothesis languages are inducable from what evidence language under what identification criterion.

It seems desirable to achieve some separation of concerns regarding the logic and control components of algorithms (or logic programs): some techniques just induce the *logic* component, assuming that the control can be added later. Adding *control* (such as by clause re-ordering inside programs and literal re-ordering inside clauses so as to ensure safety of negation-by-failure, termination, etc.) is something specific to the (idiosyncrasies of the) execution mechanism of the target language, as well as specific to the desired ways of using the induced program (which are mentioned in additional inputs, see the next sub-section). If an interpreter of the target language is actually used during the induction (say, to verify the coverage of the evidence), such control aspects cannot be entirely ignored while constructing the logic component.

A generalization of the ILP task is known as *theory-guided induction*, or (inductive) *theory revision*, or *declarative debugging*: the idea here is that an additional input is provided, namely an initial hypothesis (or: theory) H_i, under the constraint that the final hypothesis H should be as close a "variant" thereof as possible, in the sense that only the "bugs" of H_i w.r.t. E should be (incrementally) found and corrected (or: "debugged") in order to produce H. This generalized scheme reduces to the normal one in its extreme cases, that is when H_i is maximally specific or general, depending on whether induction proceeds bottom-up or top-down. In the past, this was also known as *model-driven* or *approximation-driven* learning, as opposed to *data-driven* learning, where there is no initial theory.

Another variant of the ILP task involves augmenting the inputs with *declarative bias*, which is any form of input information that restricts the search space. There are two complementary approaches to this, and we discuss them separately in the next two sub-sections.

1.1.2 Additional Specification Information

A *specification* of a program contains (*i*) a description of what problem is (to be) solved by the program, as well as (*ii*) a description of how to use the program.

The former description should define the intended relation as declaratively as possible. Whether it should be informal or formal is an on-going debate, but we don't have a choice here, since we want it to be processed by a machine. Ideally, it should even be as complete as possible, but, as mentioned earlier, this is rarely achieved in practice. The problem descriptions investigated here (the evidence) are actually even assumed-to-be-incomplete. They are furthermore the most declarative (formal) descriptions that we can imagine (if they are constrained to be non-recursive [16]).

The latter description should give the predicate symbol representing the intended relation, the sequence of names and *types* of its formal parameters, *pre-conditions* (if any) on these parameters, as well as the representation conventions of the formal parameters so that one knows how to interpret their actual values. In logic programming, where we are concerned with relations rather than functions, there should also be an enumeration of the input/output *modes* in which the program may be called (since full reversibility is rarely required or rarely even achieved in practice), as well as optional *multiplicity* (or: *determinism*) information for each mode (stating the minimum and maximum number of correct answers to a query in that mode).

Since such information is part of a (useful) specification anyway, it is only natural to provide (some of) it as an additional input to an ILP task, especially for a program synthesis task. In the ILP literature, such information is usually called *semantic bias* (a kind of declarative bias that restricts the behavior of hypotheses), but we find this terminology insufficient, as it fails to establish the link with (good) specification practice. Type and mode information are the most commonly used, and, not surprisingly, they reduce search spaces drastically. Some techniques efficiently exploit a particular case of multiplicity information, namely that the intended relation is a total function in a given mode (i.e. its multiplicity is 1–1). Of course, such statements should ideally also be provided for all the predicates defined in the background knowledge.

1.1.3 Syntactic Bias

Syntactic bias is another, complementary form of declarative bias. It restricts the language of hypotheses. Ideally, it is a parameter of an induction technique, rather than hardwired into it. As a parameter, it can be provided either by the source as an additional input, or made available to the technique by its designers.

One particularly useful and common approach is to bias induction by a schema. A *program schema* contains a template program abstracting a class of actual programs (called *instances*), in the sense that it represents their dataflow and control-flow by means of parameterized place-holders, but does not contain (all) their actual computations nor (all) their actual data structures, together with a set of *constraints* that the place-holders of the schema should satisfy.

One could for instance design a template program capturing the class of divide-and-conquer programs, or a sub-class thereof, e.g. those featuring two parameters, with division of the first parameter into two components that are somehow smaller than it:

$$\begin{array}{l} \mathsf{r(X,Y)} \leftarrow \mathsf{primitive(X),\ solve(X,Y)} \\ \mathsf{r(X,Y)} \leftarrow \mathsf{nonPrimitive(X),\ decompose(X,HX,TX_1,TX_2),} \\ \qquad\qquad\qquad \mathsf{r(TX_1,TY_1),\ r(TX_2,TY_2),\ compose(HX,TY_1,TY_2,Y)} \end{array}$$

The intended semantics (data-flow constraints) of this template can be informally described as follows. For an arbitrary relation r over formal parameters X and Y, an instance is to determine the value(s) of Y corresponding to a given value of X. Two cases arise: either X has a value (when the primitive test succeeds) for which Y can be easily directly computed (through solve), or X has a value (when the nonPrimitive test succeeds) for which Y cannot be so easily directly computed.[1] In the latter case, the divide-and-conquer principle is applied by (*i*) division (through decompose) of X into a term HX and two terms TX_1 and TX_2 that are both of the same type as X but smaller

---

[1] Note that *both* cases may apply, as there may be values of Y that it is easy to directly compute from a given X, as well as other values of Y that it is not so easy to directly compute from that X.

than X according to some well-founded relation, (*ii*) conquering (through r) in order to determine the value(s) of TY_1 and TY_2 corresponding to TX_1 and TX_2, respectively, and (*iii*) combining (through compose) terms HX, TY_1, TY_2 in order to build Y.

Enforcing this intended semantics must be done “manually,” as the template by itself has no semantics, in the sense that many programs can be seen as an instance of it, not just divide-and-conquer ones. One way of doing this is to attach to the template the set of specifications of its predicate place-holders: these specifications are in terms of each other, including the one of r, and are thus generic (because even the specification of r is unknown), but can be abduced once and for all according to the informal semantics of the schema [15]. Such a schema (i.e. template plus specification set) constitutes an extremely powerful syntactic bias, because it encodes algorithm design knowledge that would otherwise have to be hardwired or rediscovered the “hard way” during each synthesis.

There are two approaches for representing schemata. The first approach is representing the schemata as higher-order expressions, sometimes augmented by extra-logical annotations and features, where the actual programs are obtained by applying higher-order substitutions to the schema. The reason why some researchers prefer this approach is that they find this approach suitable for some applications such as schema-guided program transformation [6], where a schematic program transformation could begin only if one can find some form of higher-order matching between actual programs and schemata. In the second approach, the schemata are represented as first-order programs, where actual programs are obtained by an interpretation of the relations and the functions of the schema. In other words, the actual programs are obtained by adding programs for its *open* relations, where openness means that an arbitrary interpretation can apply to the relation and the function. This kind of schemata is called *open programs* [15]. A *synthesis strategy* determines a way in which the open relations of the schema are instantiated. There could be more than one strategy for a given schema, depending on which open relation(s) to instantiate first (e.g. instantiation of decompose, primitive, and nonPrimitive), and which open relations to instantiate next (e.g. solve and compose).

There are two ways of biasing synthesis by a schema. *Schema-based* synthesis infers a program guaranteed to fit the template of a pre-determined schema and to satisfy its specification set, but the schema itself is to a certain degree hardwired into the technique. A useful variant is *schema-guided* synthesis, where the schema is a parameter to the technique (which is thus schema-independent) and thus actively guides the synthesis. As a parameter, it can be provided either by the source as an additional input, or made available to the technique by its designers.

Less common approaches to syntactic bias are the clause description language of [1], antecedent description grammars [7], argument dependency graphs [27], etc., and are surveyed in [26].

1.1.4 Generality

Given the formula $G \Rightarrow S$, we say that G is *more general* than S, and that S is *more specific* than G. In ILP, the aim is to compute a hypothesis H given background knowledge B and evidence E, such that $\mathrm{B} \wedge H \Rightarrow E$. The generality relation $\Rightarrow$ is a partial order, but doesn't induce a lattice on the set of formulas. Indeed, there is not always a unique least generalization under implication of an arbitrary pair of clauses. For instance, the clauses $\mathsf{p(f(X))} \leftarrow \mathsf{p(X)}$ and $\mathsf{p(f(f(X)))} \leftarrow \mathsf{p(X)}$ have both $\mathsf{p(f(f(X)))} \leftarrow \mathsf{p(X)}$ and $\mathsf{p(f(X))} \leftarrow \mathsf{p(Y)}$ as least generalizations. In [22], the existence and computability of a least generalization under implication for any finite set of clauses that contains at least one non-tautologous function-free clause is proven. Since implication between Horn clauses is undecidable, there are a number of different models of inductive inference.

θ-subsumption. In the model called θ-subsumption [23], the background knowledge B is empty. The model is defined for clauses, which are viewed as sets of literals.

Definition 1.1: A clause g θ-*subsumes* a clause s iff there exists a substitution σ such that $g\sigma \subseteq s$. Two clauses are θ-*subsumption-equivalent* iff they θ-subsume each other. A clause is said to be *reduced* iff it is not θ-subsumption-equivalent to any proper subset of itself.

For instance, The clause $\mathsf{p(X,Y)} \leftarrow \mathsf{q(X,Y)}, \mathsf{r(X)}$ θ-subsumes $\mathsf{p(V,Z)} \leftarrow \mathsf{q(V,Z)}, \mathsf{q(V,T)}, \mathsf{r(V)}, \mathsf{s(Z)}$ with the substitution $\{\mathsf{X/V}, \mathsf{Y/Z}\}$.

If a clause g θ-subsumes a clause s, then $g \Rightarrow s$, but the reverse is not true for self-recursive clauses [21]. For instance, for the recursive clauses $\mathsf{p(f(X))} \leftarrow \mathsf{p(X)}$ and $\mathsf{p(f(f(X)))} \leftarrow \mathsf{p(X)}$ (called g and s respectively), although $g \Rightarrow s$ (note that s is simply g

self-resolved), g does not θ-subsume s. Therefore, θ-subsumption is not equivalent to implication among clauses. Hence, it is not adequate for handling recursive clauses.

θ-subsumption induces a lattice on the set of reduced clauses: any two clauses have a unique least upper bound (lub) and a unique greatest lower bound (glb). The least generalization under θ-subsumption (abbreviated lgθ) of two clauses c and d, denoted $lg\theta(c,d)$, is the lub of c and d in the θ-subsumption lattice. The lgθ of two terms $f(s_1,\ldots,s_n)$ and $f(t_1,\ldots,t_n)$, denoted $lg\theta(f(s_1,\ldots,s_n),f(t_1,\ldots,t_n))$, is $f(lg\theta(s_1,t_1),\ldots,lg\theta(s_n,t_n))$, whereas the lgθ of the terms $f(s_1,\ldots,s_n)$ and $g(t_1,\ldots, t_m)$, where $f \neq g$ or $n \neq m,$ is a variable V, where V represents this pair of terms throughout. The lgθ of two atoms (similarly for two negative literals) $p(s_1,\ldots,s_n)$ and $p(t_1,\ldots,t_n)$, denoted $lg\theta(p(s_1,\ldots,s_n),p(t_1,\ldots,t_n))$, is $p(lg\theta(s_1,t_1),\ldots,lg\theta(s_n,t_n))$, whereas the lgθ of the atoms $p(s_1,\ldots,s_n)$ and $q(t_1,\ldots,t_m)$, where $p \neq q$ or $n \neq m$, is T, where T denotes the “most general literal”. Finally, the lgθ of two clauses c and d, denoted $lg\theta(c,d)$, is $\{lg\theta(l_1,l_2) \mid l_1 \in c \text{ and } l_2 \in d\}$.

For instance, the lgθ of the clauses $\mathsf{p(V,W)} \leftarrow \mathsf{q(V,W), r(V), s(W)}$ and $\mathsf{p(T,N)} \leftarrow \mathsf{q(T,N), r(T), r(N)}$ is the clause $\mathsf{p(X,Y)} \leftarrow \mathsf{q(X,Y), r(X), r(Z)}$.

Relative θ-subsumption. An extension of θ-subsumption that uses background knowledge B is called relative subsumption [23].

Definition 1.2: If the background knowledge B consists of a conjunction of ground facts, then the *relative least generalization under θ-subsumption* (abbreviated rlgθ) of two ground atoms E_1 and E_2 relative to background knowledge B is $lg\theta((E_1 \leftarrow B),(E_2 \leftarrow B))$.

The rlgθ of two clauses is not necessarily finite. However, it is possible [21] to construct finite rlgθs under the syntactic bias of *ij*-determinacy.

Definition 1.3: If L_i is a literal in the ordered Horn clause $\mathsf{A} \leftarrow \mathsf{L}_1,\ldots,\mathsf{L}_n$, then the *input variables* of the literal L_i are those variables appearing in L_i that also appear in the clause $\mathsf{A} \leftarrow \mathsf{L}_1,\ldots,\mathsf{L}_{i-1}$; all other variables in L_i are called *output variables.* A literal L_i is *determinate* iff its output variables have at most one possible binding, given the binding of the input variables. If a variable V appears in the head of a clause, then the *depth* of V is zero, and otherwise, if F is the first literal containing the variable V and d is the maximal depth of the input variables of F, the depth of V is $d+1$. A clause is *ij-deter-*

minate iff it is determinate and its body contains only variables of depth at most i and predicate symbols that have arity at most j [8].

Inverse Resolution. Another model of generality is inverse resolution. There are four inductive inference rules of inverse resolution: *absorption*, *identification*, *intra-construction*, and *inter-construction* [21]:

$$\frac{(q \leftarrow A)\ (p \leftarrow A, B)}{(q \leftarrow A)\ (p \leftarrow q, B)} \qquad \frac{(p \leftarrow A, B)\ (p \leftarrow A, q)}{(q \leftarrow B)\ (p \leftarrow A, q)}$$

$$\frac{(p \leftarrow A, B)\ (p \leftarrow A, C)}{(q \leftarrow B)\ (p \leftarrow A, q)\ (q \leftarrow C)} \qquad \frac{(p \leftarrow A, B)\ (q \leftarrow A, C)}{(p \leftarrow r, B)\ (r \leftarrow A)\ (q \leftarrow r, C)}$$

In the rules above, lower-case letters represent atoms and upper-case letters represent conjunctions of atoms. The absorption and identification rules invert only one resolution step. The intra-construction and inter-construction rules introduce new predicate symbols (predicate invention, see the next subsection).

1.1.5 Predicate Invention

Predicate invention can be defined as follows: (*i*) introducing into the hypothesis some predicate(s) that are not in the evidence, nor in the background knowledge (this is called shifting the bias by extending the hypothesis language [25]), and (*ii*) inducing programs of these new predicates. This requires the usage of constructive rules of inductive inference (where the inductive consequent may involve symbol(s) that are not in the antecedent), as opposed to selective ones. Such constructive induction thus doesn't (simplistically) assume that the preliminary induction tasks of representation and vocabulary choice have already been solved, and represents thus a crucial field in induction.

One can distinguish two types of predicate invention: *necessary predicate invention* and *non-necessary predicate invention*.

Necessary Predicate Invention. We'll first give an example of necessary predicate invention, and then define it.

Example 1: In the absence of background knowledge, the induction from positive and negative examples of the following logic program for the sort predicate (where sort(L,S) holds iff S is a non-descendingly ordered permutation of L, where L, S are integer-lists):

sort([],[]) ←
sort([H|T],S) ← sort(T,Y), insert(H,Y,S)

involved the invention of the insert predicate (where insert(E,L,R) holds iff integer-list R is non-descendingly ordered integer-list L with integer E inserted), whose logic program hereafter is a by-product:

insert(E,[],[E]) ←
insert(E,[H|T],[E,H|T]) ← E≤H
insert(E,[H|T],[H|R]) ← ¬(E≤H), insert(E,T,R)

Note that the invention of the insert predicate required in turn the invention of the ≤ predicate (whose obvious specification and program are omitted here).

Definition 1.4: Predicate invention is *necessary* iff there is no finite logic program for the observational concepts in the evidence that uses only the fixed vocabulary of predicate symbols from the evidence and the background knowledge.

In Example 1, once synthesis was committed to the recursive call sort(T,Y), where T is the tail of L (i.e. L=[H|T]), the predicate insert *had to* be invented, especially that its recursive program cannot be unfolded into the program for sort. If committed to some other recursive call(s), another predicate would have had to be invented. Otherwise, the background knowledge being empty, sort would have to be implemented at most in terms of itself only, which is impossible without generating the non-terminating program sort(L,S) ← sort(L,S), or without generating an infinite program (which extensionally encodes the model).

Non-necessary Predicate Invention. One can distinguish two types of non-necessary predicate invention: *useful predicate invention* and *pragmatic predicate invention* [12].

First, we discuss useful predicate invention. If there were permutation and ordered predicates in the background knowledge of Example 1, the invention of insert such that it is recursively defined (e.g. as above) would be useful. Indeed, otherwise the insert predicate would not have to be invented as its unfoldable (because non-recursive) program would involve the permutation and ordered predicates:

insert(E,L,R) ← permutation([E|L],R), ordered(R)

and would have a complexity of O(n!), where n is the length of the list L, and would thus be inefficient compared to the recursive insert program above, which is O(n). Hence, the use of a recursive insert program would decrease the complexity of the overall sort program. The invention of a recursive insert program is thus considered useful although non-necessary.

Definition 1.5: Given a partially constructed logic program for the observational concepts in the evidence, predicate invention is *useful* iff there is a way to complete the program by inventing a predicate whose logic program is recursive.

Let's now give an example of pragmatic predicate invention.

Example 2: Given evidence of the grandDaughter relation (where grandDaughter(G,P) holds iff person G is a grand-daughter of person P), and background knowledge of the parent, female, and male relations (where parent(P,Q) holds iff person P is a parent of person Q), the induction of the following logic program for grandDaughter:

grandDaughter(G,P) ← parent(P,Q), daughter(G,Q)

involved the invention of the daughter predicate (where daughter(D,P) holds iff person D is a daughter of person P), whose logic program hereafter is a by-product:

daughter(D,P) ← parent(P,D), female(D)

The invention of the daughter predicate was pragmatic since, although the daughter program could be unfolded into the program of the grandDaughter predicate, i.e. its invention was non-necessary, inventing it caused the grandDaughter program to become more compact, and since the daughter concept has now been defined and can be reused in the future.

Definition 1.6: Given a partially constructed logic program for the observational concepts in the evidence, predicate invention is *pragmatic* iff it is neither necessary nor useful.

The task of inductive inference amounts in the limit to finding a finite axiomatization for a given model. If the intended model cannot be finitely axiomatized within a language $\mathcal{L}$, inductive inference will never succeed. However, detecting this is undecidable. This follows from Rice's theorem (see [25]):

Theorem 1: Given a recursively enumerable set of ground atoms $\mathcal{E}$ in a language $\mathcal{L}_0$, it is undecidable whether $\mathcal{E}$ is finitely axiomatizable in some language $\mathcal{L}$ such that $\mathcal{L} \supseteq \mathcal{L}_0$.

Fortunately, introducing a new predicate allows finding a finite axiomatization, as proved by Kleene (see [25]):

Theorem 2: Any recursively enumerable set of formulas in a first-order language $\mathcal{L}$ is finitely axiomatizable in the predicate calculus using additional predicate symbols not in $\mathcal{L}$.

In other words, Kleene's theorem states that inductive inference will always succeed provided the system invents the appropriate new predicates. Thus, predicate invention is crucial in inductive inference.

1.1.6 Construction Modes and Admissibility

In this sub-section, we will introduce the concepts of construction modes and admissibility [10]. The informal definitions of these two concepts are as follows: a *construction mode* for a relation states which parameter(s) are used to "construct" the other parameters, also expressing whether such usage is mandatory or optional. Construction modes should not be confused with input/output modes, which state which parameters must be ground or may be variables at call/return-time. The concept of *admissibility* captures the notion of what it means for an *atom* to satisfy a construction mode for its relation. Now, let us give the formal definitions of these new concepts. In these definitions, when we want (or need) to group several terms into a single term, we represent this as a tuple, using angled brackets. For instance, $\langle f(X,Y),g(X,Y,Z)\rangle$ is a term representing the couple built of two terms $f(X,Y)$ and $g(X,Y,Z)$.

Definition 1.7: The *leaves* of a term t, denoted $leaves(t)$, are the set of the variables and constants occurring in t.
The *vertices* of a term t, denoted $vertices(t)$, are the multi-set of the variables and function symbols (including the constants symbols) occurring in t.

For instance, $leaves(1\cdot B\cdot 1\cdot nil) = \{1, B, nil\}$, and $leaves(a\cdot T) = \{a, T\}$, whereas $vertices(1\cdot B\cdot 1\cdot nil) = \{1, \cdot, B, \cdot, 1, \cdot, nil\}$, and $vertices(a\cdot T) = \{a,\cdot,T\}$.

Definition 1.8: Term s is *syntactically obtained* from term t iff $leaves(t) \subseteq leaves(s)$. We denote this by $t \subseteq s$.

Term s *syntactically contains* term t iff $vertices(t) \subseteq_m vertices(s)$, where $\subseteq_m$ denotes multi-set inclusion. We denote this by $t \subseteq_m s$.

For instance, $\langle a,b \rangle$ is syntactically obtained from $\langle a,a \rangle$, because $leaves(\langle a,a \rangle) = \{a\} \subseteq \{a,b\} = leaves(\langle a,b \rangle)$. However, $\langle a,b \rangle$ does not syntactically contain $\langle a,a \rangle$, because $vertices(\langle a,a \rangle) = \{a,a\} \not\subseteq_m \{a,b\} = vertices(\langle a,b \rangle)$.

Definition 1.9: A construction mode m for a relation r of arity n is a total function from the set $\{1, 2,\ldots,n\}$ into the set $\{may_1,\ldots, may_n, may_{all}, must_1,\ldots, res_1,\ldots, res_n, not\}$, such that res_j is in the range of m iff may_j or $must_j$ also is in the range of m, and such that every res_j is at most once in the range of m. We also say $m(i)$ is the mode of the ith parameter of r.

A construction mode m is often written in the more suggestive form $r(m(1),\ldots,m(n))$. Do not confuse the position i of a parameter and the index j of its mode $m(i)$, say $must_j$. The intended semantics of a mode is as follows:

- mode $must_j$ means the parameter in the corresponding position is mandatory in syntactically constructing the parameter in the corresponding position of res_j;
- mode may_j means the parameter in the corresponding position is optional for syntactically constructing the parameter in the corresponding position of res_j;
- mode may_{all} means the parameter in the corresponding position is optional for syntactically constructing all other parameters;
- mode *not* means the parameter in the corresponding position is not used at all in syntactically constructing any of the parameter(s) in the corresponding position(s) of all res_j.

Let m be a mode for a relation r, and let $r(t_1,\ldots, t_n)$ be the considered atom, where n is natural number. Let the indexes in m run from 1 to k inclusive, where k is a natural number. Let $Must_j = \langle t_i \mid m(i) = must_j \rangle$, and let $Must = \langle t_i \mid m(i) = must_j \text{ for some } j \rangle$. Similarly for May_j, May_{all}, May, Res_j, Res, and Not.

For instance, let the construction mode be $arelation(may_{all}, must_1, must_2, res_1, res_2)$ and the atom be $arelation(1, [b], [], [a, b], [a])$, then we have that $k = 2$, $Must_1 = \langle [b] \rangle$, $Must_2 = \langle [] \rangle$, $Must = \langle [b], [] \rangle$, $May_1 = May_2 = \langle \rangle$, $May = May_{all} = \langle 1 \rangle$, $Res_1 = \langle [a, b] \rangle$, $Res_2 = \langle [a] \rangle$, and $Res = \langle [a, b], [a] \rangle$.

Definition 1.10: A variable is linked in a clause if it occurs in the head or if it occurs in a literal L of the body and L contains a linked variable.

Definition 1.11: A clause that has no equality atoms and no recursive calls, no T (see Definition 1.1) and no unlinked variables in the body:

$$\mathsf{r(X,Y,Z)} \leftarrow C$$

is admissible with respect to a mode *m* for *r* iff

$$\forall 1 \leq j \leq k\colon Must_j \subseteq_m \langle Res_j,\ C' \rangle \tag{1}$$

where C' is a tuple built of the atoms (seen as terms) of conjunction *C*, and

$$leaves(Res) \setminus sharedLeaves(Res) \subseteq leaves(May, May_{all}, Must, C') \cup \{0,\text{nil},\ldots\} \tag{2}$$

where *sharedLeaves*(*t*) denotes the set of leaves shared by all components of tuple *t*.

Now, we present the objective of the thesis based on the terminology and theoretical results given in Section 1.1.

1.2 The Objective of the Thesis

The learning of recursive logic programs (i.e. the class of logic programs where at least one clause is recursive, e.g. the subset program given in Section 1) from incomplete information, such as input/output examples, is a challenging subfield both of ILP (Inductive Logic Programming) and of the synthesis (in general) of logic programs from formal specifications. This is an extremely important class of logic programs, as the recent work on constructive induction [12] [25] shows that necessarily invented predicates (see Section 1.1.5) have recursive programs, and it even turns out that their induction is much harder than the one of non-recursive programs. We call this (inductive) program synthesis.

When it comes to programming applications, we believe the ideal technique is interactive (in the sense of DIALOGS [13]) and non-incremental, has a clausal evidence language plus type, mode, and multiplicity information (like SYNAPSE [11], DIALOGS), can handle semantic manipulation relations, actually uses (structured) background knowledge and a syntactic bias, which are both problem-independent and intensional (like in SYNAPSE), is guided by (and not just based on) at least the powerful divide-and-conquer schema of SYNAPSE and DIALOGS (using the implementation approach of METAINDUCE [18]), discovers additional base case and recursive case examples (like CILP [19]), can perform both necessary and useful predicate invention (like SYNAPSE, DIALOGS), even from sparse abduced evidence (like CILP), actually discovers the recursive atoms, and makes a constructive usage of the negative evidence (through abduction, like the *Constructive Interpreter* [9] and SYNAPSE).

Our aim was thus to study this important class of logic programs, i.e. recursive logic programs, and to develop a system that induces logic programs of this class. The closest system to our considerations was DIALOGS (Dialogue-based Inductive and Abductive LOgic Program Synthesizer) [13]. Therefore, we improved this system into a new one called DIALOGS-II. Thus, our aim became to improve DIALOGS, whose ancestor was the SYNAPSE system [11] [14], which induces recursive logic programs from a set of positive examples, and a set of Horn clauses that are called *properties.* The drawbacks of SYNAPSE are that the specifier may not always provide properties that are needed to induce a logic program that is correct with respect to its specification, and that most positive examples are redundant with the properties.

DIALOGS-II is a schema-guided, interactive, and non-incremental synthesizer of recursive logic programs that takes the initiative and queries a (possibly naive) specifier for evidence in her/his conceptual language. DIALOGS-II needs no properties, and only asks for the minimal knowledge a specifier *must* have in order to want a (logic) program, and it can be used by any learner (including itself) that detects, or merely conjectures, the necessity of invention of a new predicate. Moreover, due to its powerful codification of "recursion-theory" into program schemata and schematic constraints, it needs very little evidence and is very fast.

DIALOGS-II is schema-guided. The reason why it is schema-guided is as follows: most (but not all) inductive/abductive synthesizers require large amounts of ground positive and negative examples of the intended concept. This is because ground examples are not an adequate way of communicating a concept to a computer and/or because the underlying "recursion theory" of the synthesizer is poor. In order to overcome this deficiency, some researchers used non-ground examples [20], or Horn clauses [11] [14] as evidence language instead of using only ground examples, and some experimented with schema-based synthesis [11] [14] to address the poor "recursion theory" problem [17]. We chose the schema-guided approach, because we think that it is the best approach to handle "recursion theory". The schemata of DIALOGS-II are open programs and are available to the system together with their synthesis strategies. In other words, for a particular synthesis, a schema together with a synthesis strategy is chosen.

DIALOGS-II can be used to synthesize programs by making use of the available schemata and strategies that are already existing in the system. Moreover, the specifier can provide additional schemata using the declarative syntax of the schemas of the system to encode new schemata, and adding the code for strategies for those new schemata. In that way, the specifier can make syntheses of programs by executing the strategies that fit to the schemata added.

DIALOGS-II is interactive, because the specifier is assumed to be "lazy" in the sense that s/he is reluctant to take the initiative and type in evidence of the intended concept without knowing whether it will be "useful" to the synthesizer or not [13]. Therefore, DIALOGS-II takes the initiative and queries the specifier only for strictly necessary evidence. The query and answer languages are carefully designed so that even a computationally naive specifier can use the system. Moreover, it is guaranteed that the specifier can answer such queries, because otherwise the specifier would not need the synthesized program.

DIALOGS-II is a system that only induces recursive logic programs because we believe that inducing recursive logic programs is important [12], especially that they are strictly necessary (see Section 1.1.5).

DIALOGS-II is a recursive synthesizer, which means it recursively calls itself when a necessary predicate invention is conjectured during the synthesis. It is then a natural solution for the system to call itself recursively to make this new synthesis since the problem (of synthesizing a program for a necessary new predicate) has the same nature as the problem of synthesizing a program for the top-level predicate. That is, for both cases, the necessity of predicate invention is conjectured before starting a synthesis.

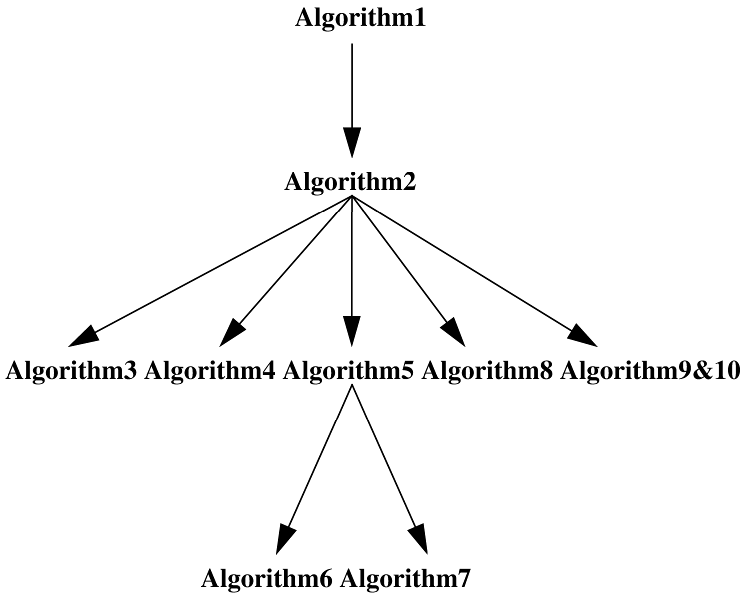
DIALOGS-II is non-incremental, because we believe that using an incremental approach is not practical for program synthesis [17]. Recursive programs are so fragile objects that they should be handled with utmost care. Therefore, we believe that using general-purpose induction techniques to synthesize programs by incrementally "debugging" the empty program (or an approximate program) according to incomplete evidence is not an appropriate way of synthesizing programs. Moreover, in incremental synthesis, the order of the evidence is important. That means the system can be forced into the synthesis of infinite, redundant, or dead code. We strongly believe that the only way to reliably and efficiently synthesize recursive programs from incomplete information is through guidance by a schema capturing a design methodology (e.g. a divide-and-conquer schema), as well as through non-incremental handling of the evidence.

In the remainder of this thesis, we will examine the DIALOGS-II technique closely in Chapter 2. This will be followed by a comparison of DIALOGS-II with current ILP systems in Chapter 3, and finally, we reach a conclusion in Chapter 4.

Chapter 2

The DIALOGS-II Technique

As mentioned earlier, DIALOGS-II is a schema-guided, interactive, recursive, and non-incremental recursion synthesizer that takes the initiative and queries a (possibly computationally naive) specifier for evidence in her/his conceptual language. In the following sub-sections, we will illustrate how the DIALOGS-II mechanism works by means of sample syntheses. First, we illustrate the synthesis of a program for the delOdds(L,R) predicate, where delOdds(L,R) holds iff R is L without its odd elements, where L, R are integer-lists. Next, we examine the synthesis of a program for the predicate reverse(L,R), where reverse(L,R) holds iff list R is the reverse of list L, to illustrate the recursive call of DIALOGS-II to itself. Before giving the sample syntheses, we give an algorithm call chart of the basic synthesis algorithm of how DIALOGS-II works and the basic synthesis algorithm itself:



Algorithm 1: schemaGuidedDialogs-II(Pgm)

```
Inputs: (none)
Outputs: Pgm
ask for the predicate declaration of the predicate for which a program is being
synthesized
PredDecl := ask('Predicate Declaration')
ask for a schema and a strategy for the schema
selectSchemaStrategy(Schema,Strategy)
call Dialogs-II with Schema, Strategy and PredDecl to induce Pgm
dialogsII(Schema,Strategy,PredDecl,Pgm)
```

As shown in Algorithm 1, after executing the first two statements, the system executes the statement `dialogsII(Schema,Strategy,PredDecl,Pgm)` whose algorithm is given as follows:

Algorithm 2: dialogsII(Schema,Strategy,PredDecl,Pgm)

```
Inputs: Schema, Strategy, PredDecl
Outputs: Pgm
execute the strategy in order to obtain an open program from the schema, where
the open program has open relations to be "closed" by the end of the next two
statements (i.e. abduce and induce). ParamRoles denotes the information about
```

```
the names, types, and roles of the parameters (e.g. induction, result).
Strategy(PredDecl,Schema,OpenPgm,ParamRoles)
abduce the evidence necessary for "closing" the open relations p and q of the open program by means of querying the specifier, where the open relation of the non-recursive clause is p, whereas the open relation of the recursive clause is q. The atoms of these relations are supposed to be the last atoms of the non-recursive and recursive clauses of the open program respectively.
abduce(OpenPgm,ParamRoles,PredDecl,pEvidence,qEvidence)
induce the programs for the open relations by using the Program Closing Method based on the evidence abduced in the previous step according to the construction modes pMode and qMode of the relations p and q respectively.
induce(pEvidence,qEvidence,pMode,qMode,pClauses,qClauses)
evaluate the result of the Program Closing Method to conjecture if there is a need for inventing a new predicate
evaluate(Schema,Strategy,OpenPgm,pClauses,qClauses,PredDecl,ParamRoles,Pgm)
```

Now, we go through the statements of the basic synthesis algorithm (Algorithm 1) for the synthesis of a program for the `delOdds(L,R)` predicate. We will first discuss the first two statements of this algorithm: asking for a predicate declaration, selecting a schema and a strategy in Section 2.1. Next we will go through the statements of Algorithm 2 by first discussing the execution of the strategy in Section 2.3 and abduction of evidence in Section 2.4, which is followed by the discussion of the induction of program clauses in Section 2.5, and finally by the evaluation of the program clauses to conjecture necessary predicate invention and sparseness problem in Section 2.5.2.

2.1 Asking For a Predicate Declaration, a Schema and a Strategy

DIALOGS-II first needs to know for which predicate it is synthesizing a program. Therefore, it asks the predicate declaration of the predicate. The specifier *must* be able to give such a declaration, because otherwise s/he would not have the need to have a program for this predicate. Thus, the first step in the synthesis is prompting the specifier for a predicate declaration and obtaining it:

`Predicate declaration?` delOdds(L:list(int),R:list(int))

where the type of L and R is list(int). Other available types are in the set {atom, int, nat, list(_),...}.

As mentioned earlier, DIALOGS-II is a schema-guided synthesizer. Therefore, it needs a schema and a strategy for the schema in order to be able to start a synthesis. Thus, the next step in the synthesis is prompting the specifier for a schema and a strategy for this schema.

A basic algorithm for selecting a schema and a strategy for it is given below, where SchemaDefaults is a parameter representing the list of available schemata in the system, Schema is a schema in SchemaDefaults, and Strategy is a strategy for Schema.

```
Algorithm 3: selectSchemaStrategy(Schema,Strategy)
Inputs: none
Outputs: Schema, Strategy
ask the specifier to select Schema from SchemaDefaults in the system
Schema := ask('Schema', SchemaDefaults)
determine StrategyDefaults, the list of available strategies for Schema
StrategyDefaults := determineStrategyDefaults(Schema)
ask the specifier to select Strategy from StrategyDefaults
Strategy := ask('Strategy', StrategyDefaults)
```

Now, let us see how is this done during the synthesis of a program for delOdds(L,R). Note that the questions of this dialog are in the `typewriter` font, the specifier's answers are in helvetica font, and the default answers of the system are given inside curly braces, i.e. {}, and suppose that one of the schemata available in the system is a "divide-and-conquer" schema together with a strategy for it:

`Schema? {divide-and-conquer1}` divide-and-conquer1

`Strategy? {divide-and-conquer-Strategy1}` divide-and-conquer-Strategy1

Now, DIALOGS-II knows that it will use a divide-and-conquer schema with a particular strategy, i.e. *divide-and-conquer-Strategy1*.

2.2 Execution of the Strategy

The next step is to execute the strategy selected by the specifier. Before giving the algorithm of a particular strategy, let us see what the considered divide-and-conquer schema looks like. The considered schema is:

$$\mathsf{r}(\mathsf{X},\mathsf{Y},\mathbf{Z}) \leftarrow \mathsf{solve\_r}(\mathsf{X},\mathsf{Y},\mathbf{Z})$$
$$\mathsf{r}(\mathsf{X},\mathsf{Y},\mathbf{Z}) \leftarrow \mathsf{decompose\_r}(\mathsf{X},\mathbf{HX},\mathbf{TX}),\ \mathsf{r}(\mathsf{TX}_1,\mathsf{TY}_1,\mathbf{Z}),\ldots,\mathsf{r}(\mathsf{TX}_t,\mathsf{TY}_t,\mathbf{Z}),$$
$$\mathsf{compose\_r}(\mathbf{HX},\mathbf{TY},\mathsf{Y},\mathbf{Z})$$

where $\mathbf{HX}=\mathsf{HX}_1,\ldots,\mathsf{HX}_h$, $\mathbf{TX}=\mathsf{TX}_1,\ldots,\mathsf{TX}_t$, $\mathbf{TY}=\mathsf{TY}_1,\ldots,\mathsf{TY}_t$, and $\mathbf{Z}=\mathsf{Z}_1,\ldots,\mathsf{Z}_z$.

A *divide-and-conquer* program for a predicate r over parameters X, Y, and **Z** works as follows. Suppose that X is the induction parameter, Y is the result parameter, and **Z** the (repetitive) passive parameter(s), where a *passive* parameter is a parameter that does not change through a recursive call. There are two possibilities of how Y can be computed: the first one is that Y is directly computed from X and **Z** by means of solve_r(X,Y,**Z**). There could be more than one way in which Y is directly computed from X and **Z** (in other words, there could be more than one clause whose head is solve_r(X,Y,**Z**) in the final synthesized program). In the second one, first X is decomposed into *h* heads and *t* tails by means of decompose_r(X,**HX**,**TX**). Next, *t* recursive calls are done, one for each TX_i. Last, the result parameter Y is constructed from **HX**, **TY**, and **Z** by means of compose_r(**HX**,**TY**,Y,**Z**). To be precise, the **HX** are processed and composed with the **TY** and **Z** in order to yield Y. Again, there could be more than one way of computing Y from **HX**, **TY,** and **Z**. The schema given above is a representation of this algorithm description.

So, in order to generate an open program from this schema according to the strategy *divide-and-conquer-Strategy1*, the system must determine and use the roles of the parameters, the number of passive parameter(s) (if any), i.e. $0 \leq z$, the program for the open relation decompose_r, and *h* and *t*.

Now, let us give the algorithm for executing the strategy for the divide-and-conquer schema given above:

```
Algorithm 4: divide-and-conquer-Strategy1(PredDecl,Schema,Pgm,ParamRoles)
Input: PredDecl, Schema
Output: Pgm, ParamRoles
determine the induction parameter, which is of an inductively defined type, the result parameter (if any), and the passive parameter(s) (if any), and the number of result and passive parameters, i.e. $y$ and $z$ respectively, from the predicate declaration PredDecl
⟨ParamRoles,$y$,$z$⟩ := paramRoles(PredDecl)
determine decompose using the system-defined decomposition operators, i.e. DecomposeDefaults
⟨decompose_r,$h$,$t$⟩ := selectDecompose(DecomposeDefaults)
Pgm := generateOpenPgm(Schema,decompose_r,$h$,$t$,$z$)
```

Now, we examine the execution of the strategy *divide-and-conquer-Strategy1* by means of the synthesis of a program for delOdds(L,R).

First, we show the determination of the parameter roles using the predicate declaration *delOdds*(*L*:*list*(*int*),*R*:*list*(*int*)): DIALOGS-II creates a sequence of potential induction parameters, which are of inductively defined types, keeps the first one as the (first) default answer, and the remaining ones as default ones upon backtracking. Similarly for the result parameter, which is also likely to be of an inductively defined type: from the currently remaining parameters, DIALOGS-II can create a sequence of potential result parameters, keep the first one as the (first) default answer, and the remaining ones as default answers upon backtracking. Finally, DIALOGS-II can propose as the passive parameter(s) (if any) the remaining parameter(s) (if any). Providing default answers is good for naive specifiers, where naive specifiers are the ones who do not have the capability for answering every question of the system, since if s/he has no idea of determining the roles of the parameters, s/he can simply accept the default answers and go on with the synthesis without blocking at this step. Note that a passive parameter may accidentally be declared as a result parameter, without any influence on the existence of a correct program: it would be found to be always equal to its tail by post-synthesis transformations, where in that case the synthesis would be a bit slower, because unnecessary computations would need to be done for its construction using its tail, **HX**, and the actually declared passive parameters.

How the parameter roles of delOdds(L,R) are determined is shown by the dialogue below, supposing that the specifier accepts the default answers proposed by the system. First, the specifier is prompted for the induction parameter, where the system proposes the parameter L as the default answer:

```
Induction parameter? {L} L
```

Next, the specifier is prompted for the result parameter, where the system proposes the (remaining) parameter R as the result parameter since there is only one remaining parameter according to the predicate declaration and it has to be a result parameter since the result parameter is asked before passive parameters.

```
Result parameter? {R} R
```

Note that there is (are) no passive parameter(s).

The strategy selected by the specifier makes DIALOGS-II create a sequence of potential decomposition operators using available decomposition operators in the system, keep the first one as the (first) default answer, and the remaining ones as default ones upon backtracking. The specifier can select the default one or can write her/his own decompose_delOdds as an answer, where the predicates in the body must already be defined as procedures in the system; let us assume that the specifier selects the default one, which is a head-tail decomposition of the list:

```
Decomposition operator? {decompose_delOdds(L,HL,TL)←
L=[HL|TL]}
decompose_delOdds(L,HL,TL) ← L=[HL|TL]
```

The other pre-defined decomposition operators of the type list(_) are given below, where *h* denotes the number of heads and *t* denotes the number of tails:

decompose_r(L,H_1,H_2,T) ← L=[H_1,H_2|T] *h*/2, *t*/1

…

decompose_r(L,H,T_1,T_2) ← L=[H|T], partition(T,H,T_1,T_2) *h*/1, *t*/2

decompose_r(L,T_1,T_2) ← L=[_,_|_], halves(L,T_1,T_2) *h*/0, *t*/2

…

Similar sequences are also available for other inductively defined types, e.g. nat. Next, *h* and *t* are instantiated according to the selected decomposition operator: for head-tail decomposition, both *h* and *t* are 1. At this time of the synthesis, from a programming point of view, all creative decisions have been taken, but alternative decisions are ready for any occurrence of backtracking (either because DIALOGS-II fails due to some decision at a later step of Algorithm 2, or because the specifier wants another program after successful completion of all the steps).

Knowing decompose_delOdds, and the values of *h*, *t*, *z*, the following open program for delOdds(L,R) is generated from the input schema:

```
delOdds(A,B) ← solve_delOdds(A,B)
delOdds(A,B) ← decompose_delOdds(A,C,D), delOdds(D,E),
                                      compose_delOdds(C,E,B)
decompose_delOdds(F,G,H) ← F=[G|H]
```

Note that the relations solve_delOdds and compose_delOdds are open: they will be "closed" after the execution of the second and the third statements (abduction of evidence and induction of clauses) of Algorithm 2. This open program is passed as an input to the second statement of Algorithm 2.

2.3 Abduction of Evidence for the Open Relations of the Open Program

Let the open relations of an open program be p and q, where p is the open relation of a non-recursive clause and q is the open relation of a recursive clause of the open program.

In DIALOGS-II, the process of finding programs for the open relations p and q is also interactive and is based on the notions of abduction through (naive) unfolding and querying, and induction through the Program Closing Method (computation of least general generalizations).

We will illustrate naive unfolding and querying by means of the open relations of an open program of the divide-and-conquer schema given previously. The basic principle of (naive) unfolding and querying is as follows. Based on an open program

```
r(A,B) ← solve_r(A,B)
r(A,B) ← decompose_r(A,C,D), r(D,E), compose_r(C,E,B)
decompose_r(F,G,H) ← F=[G|H]
```

whose induction parameter is A, result parameter is B, decomposition operator is a head-tail one, and open relations are solve_r and compose_r (where solve_r denotes p and compose_r denotes q, respectively), the possible computation “traces” for various most general values of the induction parameter are:

$r([],D_1) \leftarrow solve\_r([],D_1)$
$r([E_1],F_1) \leftarrow solve\_r([E_1],F_1)$
$r([E_1],F_1) \leftarrow r([],F_2), compose\_r(E_1,F_2,F_1)$
$r([G_1,G_2],H_1) \leftarrow solve\_r([G_1,G_2],H_1)$
$r([G_1,G_2],H_1) \leftarrow r([G_2],H_2), compose\_r(G_1,H_2,H_1)$
…

The basic principle is to (*i*) query the specifier for an instance of the last atom of each trace, using previous answers to resolve recursive calls, (*ii*) induce a program for solve_r from some of the answers so that it is not an open relation afterwards, (*iii*) induce a program for compose_r from the other answers so that it is not an open relation after this induction. The criterion of how to make such a division of the answers follows from the construction modes (see Section 1.1.6) of the schema. Before giving the steps above in detail, we introduce a new concept.

Definition 2.1: (Most general form of a parameter)
The most general form of a parameter of a certain type *t* and of a certain size *s* is denoted by

$$mostGenForm(t,s)$$

and is found using type-specific programs. For instance, for type list, the program is as follows:

```
list(nil,0) ←
list(H.T,M) ← list(T,N), M is N+1
```

The most general form X of a parameter of type list and of size 3 is computed by SLD resolution of the goal

```
← list(X,3)
```

with the program given above yielding the list A.B.C.nil. Similarly, for type nat, the program is

```
nat(0,0) ←
nat(s(N),M) ← nat(N,T), M is N+1
```

The most general form X of a parameter of type nat and of size 2 is computed by SLD resolution of the goal

$\leftarrow$ nat(X,2)

with the program given above yielding the natural number s(s(0)).

Step (*i*) is realized by means of a basic loop: for each most general form of the induction parameter a goal for the top-level predicate is generated. For each clause whose head unifies with that goal, the atom of an open relation in the body of the clause is found by resolving the body atoms ("executing" the body) using the primitives, specifier-introduced predicates (which are introduced while the specifier gives answers to the queries about the predicate for which a program is being synthesized), and the clauses abduced during the previous iterations of the loop. And for each such an "open" atom, a query is generated. From the specifier's answer to the query, some evidence is abduced for the open relation. This basic loop is repeated until the user answers a query by *stop-it*.

Let us now give an algorithm for Step (*i*) (note that Step (*ii*) and Step (*iii*) will be discussed in the following sub-sections). The algorithm abduces evidence, i.e. pEvidence and qEvidence, for the open relations p and q, where Pgm is the open program, ParamRoles is information about the parameters of Pgm, i.e. names, types, and the positions of the parameters in the heads of the clauses of Pgm, which is computed using the predicate declaration PredDecl by Algorithm 4, and TopPred is the name of the predicate for which a program is being induced.

```
Algorithm5: abduce(Pgm,ParamRoles,PredDecl,pEvidence,qEvidence)
Inputs: Pgm, ParamRoles, PredDecl
Outputs: pEvidence, qEvidence
Shortcuts are abduced clauses for the open relations p, q and for TopPred
Shortcuts := {}
pEvidence := {}
qEvidence := {}
$i$ := 0
repeat
    let $X_i$ be the most general form of the induction parameter of type t of size i
    $X_i$ := mostGenForm(t,i)
    construct a goal using $X_i$ and variable result and passive parameter(s)
    TopPred := predName(PredDecl)
    Goal := TopPred($X_i$,Y,Z)
    find a clause (in Pgm) whose head unifies with Goal and whose body unifies
    with Body (under the same substitution)
    Body := pgmClause(Pgm,Goal)
    prove Body in order to find an atom of open relation p or q
    demo(Body,Pgm,TopPred,Shortcuts,Background,Assumptions,
    ResidueAtom)
    query the specifier to abduce evidence for the open relation of Body
    askQuery(Goal,ResidueAtom,Assumptions,Answer)
    if Answer ≠ “false” and Answer ≠ “stop-it” then
      abduce evidence for open relations p or q using the answer Answer to the
      query made in askQuery
      if ResidueAtom is of predicate p then
        ⟨pExs,Shortcut⟩ := abduceClauses(Answer,ResidueAtom)
        assert pExs and Shortcut
        pEvidence := pEvidence ∪ pExs
      else
        ⟨qExs,Shortcut⟩ := abduceClauses(Answer,ResidueAtom)
        assert qExs and Shortcut
        qEvidence := qEvidence ∪ qExs
      Shortcuts := Shortcuts ∪ {Shortcut}
    else
      abduce nothing
    increment i
  until Answer = “stop-it”
  retract all Shortcuts to prevent them being used for further syntheses
```

Now let us give the algorithm for *demo*:

```
Algorithm 6: demo(Goal,Pgm,TopPred,Shortcuts,Background,
Assumptions,ResidueAtom)
Input: Goal, Pgm, TopPred, Shortcuts, Background
Output: Assumptions, ResidueAtom
$(\mathsf{Pgm} + \mathsf{Shortcuts}) \cup \mathsf{Background} \cup \mathsf{Assumptions} \cup \mathsf{ResidueAtom} \vdash_{\mathsf{SLD}}$
Goal
```

Let us explain how *demo* works now: the proof of Goal is done by using Shortcuts and Background. Shortcuts are abduced clauses for the open relations p, q and for TopPred, where these clauses have precedence over the clauses of during SLD resolution (note that abduced clauses for the open relations p, q are also called evidence since they will be used as evidence for closing these open relations). That is, when the head of a shortcut clause unifies with an atom in Goal, these shortcut clauses are used instead of the clauses of Pgm (note that + is used instead of ∪ to indicate this precedence in Algorithm 6). If there is neither a shortcut for an atom nor a clause in Pgm whose head unifies with that atom, then resolution is impossible and the resolution of Goal stops there, where this atom is ResidueAtom. The resolution of Goal also stops when Goal is proved to be *true*. Background is a set of programs for pre-defined primitives, such as "=", ">", etc. The atoms of specifier-introduced predicates (introduced by the answers that the specifier gives to the queries) encountered in Goal are called Assumptions, meaning that these atoms are assumed to be true during the SLD resolution since these atoms are introduced by the specifier and, thus, there is not any program for the specifier-introduced predicates that is known to the system, which implies that the resolution will be blocked by the atoms of the specifier-introduced predicates, if they are not assumed to be true. Assumptions are collected (through conjunction) in order to be passed to the query-asking during which the query is designed by considering Assumptions to be true (see askQuery below). Now, we give the algorithm for asking queries:

```
Algorithm 7: askQuery(Goal,ResidueAtom,Assumptions,Answer)
Inputs: Goal, ResidueAtom, Assumptions
Output: Answer
if ResidueAtom is true then
    do not query the specifier, because there is no atom for which any evidence
    should be abduced, thus Answer is an empty set
```

```
        Answer := {}
    else if ResidueAtom is an atom of the relation p or q then
        if Assumptions = [ ] then
            ask the query: "When does Goal hold?" and get Answer from the specifier
            Answer := ask('When does' Goal 'hold?')
        else
            ask the query: "When does Goal hold, assuming Assumptions?" and get
            Answer from the specifier
            Answer := ask('When does' Goal 'hold assuming' Assumptions?)
```

Now we know how the evidence is abduced for the open relations of an open program. Let us now examine how the abduction of evidence for the open relations solve_delOdds and compose_delOdds is done during the synthesis of a program for the delOdds predicate by considering the open program

```
delOdds(A,B) ← solve_delOdds(A,B)
delOdds(A,B) ← decompose_delOdds(A,C,D), delOdds(D,E),
                                             compose_delOdds(C,E,B)
decompose_delOdds(F,G,H) ← F=[G|H]
```

and considering that the relation solve_delOdds plays the role of the relation p, and the relation compose_delOdds plays the role of the relation q. This correspondence of the relations is due to the fact that solve_delOdds is the open relation of the non-recursive clause of the open program, and compose_delOdds is the open relation of the recursive clause of the open program.

First Iteration for Abducing Evidence. The specifier must know the value of the result parameter when the induction parameter is the empty list, otherwise s/he would not have the need for a program for delOdds. Thus, the first most general form of the induction parameter A is [], where the query generation process proceeds by first resolving the goal delOdds([],B) with the head of the recursive clause of the open program and finding a goal for resolution. But, this attempt fails after resolving decompose with the recursive clause since the induction parameter has a value, i.e. [ ], that cannot be decomposed. Therefore, the non-recursive clause is considered next. The recursive clause of the open program is tried first, because in that way the answers that the specifier gives to the queries are shorter (thus it is less boring for the specifier to answer the queries) than in the case where the non-recursive clause is used first. This is because during the resolution of a goal that has been generated by resolving the goal with the head of the recursive clause, more assumptions are likely to be collected to be passed to the queries than in the case where the goal is resolved with the non-recursive clause. More assumptions during the querying causes the specifier to write less conditions in order to make the goal (the one that includes a most general form of the induction parameter) hold.

Thus, next the goal delOdds([ ],B) is resolved with the non-recursive clause of the open program yielding the goal:

← solve_delOdds(A,B)

Resolving this goal is impossible, so the unfolding process stops here, and DIALOGS-II extracts the following query to abduce evidence for solve_delOdds:

```
When does delOdds([],B) hold?
```

from this goal (see Algorithm 7). Note that the specifier should be able to answer this query, since otherwise s/he would not need a program for the predicate delOdds, in that sense the specifier is guaranteed to answer the queries. The answer should be a formula $\mathcal{F}$[B], where only B may be free, explaining how to compute B from [] such that delOdds([],B) holds. In other words, solve_delOdds([ ],B) should be "equivalent" to $\mathcal{F}$[B]. The answer to the query is: *B*=[]. Using this answer, DIALOGS-II abduces the following evidence and shortcuts for solve_delOdds and delOdds (see Algorithm 5):

solve_delOdds([],A) ← A=[]
delOdds([],A) ← A=[] (s1)

Second Iteration for Abducing Evidence. The specifier must also know the result when the induction parameter is a one-element list. The query generation process starts by unifying the goal delOdds([A],B) with the head of the recursive clause of the open program yielding the goal:

← decompose_delOdds(A,C,D), delOdds(D,E),
compose_delOdds(C,E,B)

Resolving decompose_delOdds(A,C,D) and resolving the resulting equality atom gives

← delOdds([],E), compose_delOdds(C,E,B)

Using the shortcut s1 and resolving the resulting equality atom yields:

← compose_delOdds(C,[],B)

Now the following query can be extracted from this goal since resolving this goal is impossible. The specifier answers the query as follows (note that the comma "," stands for conjunction, and the semi-colon ";" stands for disjunction, where the comma has a higher precedence than the semi-colon):

`When does delOdds([A],B) hold?` B=[], odd(A); B=[A], even(A).

Note that the predicates *odd* and *even* are introduced by the specifier, where the atoms *odd*(*X*) and *even*(*X*) are from now on assumed by the system to be true. Otherwise, resolving these atoms would be impossible and the resolution will be blocked because there are no programs for the predicates *odd* and *even*. Instead of blocking when such atoms are encountered, the system keeps these atoms to pass them to the queries (see the third iteration for abducing evidence given below). Using this answer to the query, DIALOGS-II abduces the following evidence and shortcuts (note the correspondence between the answers in the answer disjunct and the bodies of the shortcut and evidence clauses):

compose_delOdds(A,[],B) ← B=[], odd(A) (s2)
compose_delOdds(A,[],B) ← B=[A], even(A) (s3)
delOdds([A],B) ← B=[], odd(A) (s4)
delOdds([A],B) ← B=[A], even(A) (s5)

Now, upon backtracking, unifying the goal delOdds([A],B) with the head of the non-recursive clause of the open program yields the goal:

← solve_delOdds([A],B)

where resolving this goal is impossible. In this case, DIALOGS-II directly collects evidence for solve_delOdds using the shortcuts s4 and s5 instead of generating a query that would be identical to the one made for the abduction of evidence for compose_delOdds(C,[ ],B). Thus, the evidence collected for solve_delOdds is the following:

solve_delOdds([A],B) ← B=[], odd(A)
solve_delOdds([A],B) ← B=[A], even(A)

Third Iteration for Abducing Evidence. Next, the specifier is queried for the result when the induction parameter is a two-element list. Again, the specifier *must* know the answer. DIALOGS-II first creates the following clause by unifying the goal delOdds([A,B],C) with the head of the recursive clause of the open program yielding the goal:

$\leftarrow$ decompose_delOdds([A,B],HA,TA), delOdds(TA,TB),
compose_delOdds(HA,TB,C)

Resolving decompose_delOdds([A,B],HA,TA) and the resulting equality atom, and using the shortcut s4 reduces this goal to:

$\leftarrow$ odd(B), compose_delOdds(A,[],C)

Note that the atom odd(B) is an atom of the specifier-introduced predicate odd, and remember that during the SLD resolution of a goal, if such an atom is encountered, then this atom is assumed to be true since it was introduced by the specifier, and kept since it is passed to the next query. Thus, the goal becomes:

$\leftarrow$ compose_delOdds(A,[],C)

Using s2, this becomes:

$\leftarrow$ odd(A), C=[]

Again note that odd(A) is assumed to be true since it is an atom of the specifier-introduced predicate, and it is kept for the next query. So, now Assumptions becomes equal to the set {odd(B), odd(A)}. Thus, the goal becomes:

$\leftarrow$ C=[]

which is finally resolved to:

$\leftarrow$ true

Since there is no atom of any open relation in that goal, no query can be generated from it (thus, in that case the assumptions collected are not used).

Next, upon backtracking, by the use of the other shortcut, i.e. s5, the following goal is obtained:

$\leftarrow$ even(B), compose_delOdds(A,[B],C)

where the atom even(B) is assumed to be true and collected as an assumption to be passed to the next query, again because it is an atom of a specifier-introduced predicate. Thus, the goal becomes:

← compose_delOdds(A,[B],C)

where resolving this goal is impossible, so that the following query is generated (note the usage of the assumption even(B) in the query):

`When does delOdds([A,B],C) hold, assuming even(B)?`

C =[B], odd(A); C=[A,B], even(A).

The following shortcuts and evidence are abduced from the answer:

compose_delOdds(A,[B],C) ← C=[B], odd(A)
compose_delOdds(A,[B],C) ← C=[A,B], even(A)
delOdds([A,B],C) ← C=[B], odd(A), even(B) (s6)
delOdds([A,B],C) ← C=[A,B], even(A), even(B) (s7)

Unifying the goal delOdds([A,B],C) with the head of the non-recursive clause of the open program would yield the goal

← solve_delOdds([A,B],C)

Since the system now knows when delOdds([A,B],C) holds (see shortcuts s6 and s7), the specifier is not queried, and by using the shortcuts s6 and s7, DIALOGS-II directly abduces the evidence:

solve_delOdds([A,B],C) ← C=[B], odd(A), even(B)
solve_delOdds([A,B],C) ← C=[A,B], even(A), even(B)

If first the goal delOdds([A,B],C) had been unified with the non-recursive clause yielding the goal

← solve_delOdds([A,B],C)

where resolving this goal is impossible, then the specifier would have been queried as follows:

`When does delOdds([A,B],C) hold?`

where s/he should have answered this query as:

C =[B], odd(A), even(B); C=[A,B], even(A), even(B); C=[], odd(A), odd(B); C=[A], even(A), odd(B)

Note that the specifier would have to write a longer answer for this query than for the one that was asked for compose_delOdds. That is why the goal is unified first with the recursive clause rather than the non-recursive one as explained earlier.

Stopping the Query Session. Next, the specifier is queried for the result when the induction parameter is a three-element list. Suppose that the specifier is bored or believes having said sufficiently many useful things about delOdds and does not want to answer any queries anymore. In that case, the specifier answers the query by the keyword "stop-it", so that the query session is ended:

```
When does delOdds([A,B,C],D) hold, assuming even(B),
even(C)? stop-it.
```

Stopping the querying is thus fully manual (specifier-dependent). Actually, there are two other possibilities to stop querying: the first one is fully automatic, the second one is semi-automatic.

In the first one, a heuristic is used to conjecture whether the system has to stop querying or not. The heuristic is as follows: after abducing evidence for p and q after each query, all the abduced evidence for p and q is processed (by the Program Closing Method) and compared with the result of the same process done on the evidence collected for the previous query. If the results of these two processes are the same, then it is assumed that the potential next queries would also yield the same results, so it is conjectured that the system can stop querying and rely on the evidence that was collected until that time. This method is fully-automatic, because the system makes its decision without any interaction with the specifier. But, due to its being a heuristic, the system can be defeated.

The second method is a combination of the other two methods. The system processes all the evidence after each query, and if the last two successive results are the same, it asks the specifier to conjecture whether to continue querying or not, since there is a possibility that the abduced evidence is adequate for induction of a correct program. If the specifier thinks that this much evidence is sufficient to induce a correct program, then a program is induced from this evidence, otherwise s/he is further queried until s/he decides that the abduced evidence is adequate.

We think that the DIALOGS-II method is the most appropriate one. Its method is better than the fully-automatic one since it leaves the decision to the specifier, so that it is always possible to induce a correct program either by a first correct decision of the specifier on stopping querying, or by successive syntheses that would let the specifier synthesize a correct program in the end, by making the specifier learn that s/he should answer more queries each time the system is re-run. This method has a drawback because of its being a heuristic. It fails when a correct program can only be induced after some other queries. That is, abduction of some more new evidence could cause a change in the result of each process done after each query. In that case, the program induced could be incomplete/incorrect. The second method is mostly for expert specifiers since the decision whether the abduced evidence is adequate or not is not an easy decision for a naive specifier, where a specifier who has the knowledge and capability to make such a decision is considered an expert specifier, whereas a specifier who is not capable of making such a decision is considered a naive one. However, the naive specifier could decide to stop querying the first time the system asks to make a decision. In that case, this method boils down to a combination of the other two methods.

Now, let us see how the abduced evidence for solve_delOdds and compose_delOdds will be processed in order to find programs for these relations.

2.4 Induction of Clauses: The Program Closing Method

The Program Closing Method discussed in this section is based on the Program Closing Method discussed in [10]. There, the open program has only one relation that will be closed using evidence for that relation. According to our Program Closing Method, there are two open relations of the open program, i.e. p and q. Let us see now how it works.

The evidence abduced for the open relations p and q during the execution of the third statement of Algorithm 2 is divided into subsets such that the lgθ of each subset yields a clause for either p or q. In order to understand how this division into subsets and taking the lgθ of each subset is done, we have to first analyze the dataflow of the programs that have p and q as open relations. In other words, we have to look inside the open relations p and q.

Here, we analyze the data-flow of divide-and-conquer programs, which have solve_r and compose_r as open relations (see the divide-and-conquer schema on page 23).

Using general knowledge of the divide-and-conquer design methodology, it is possible to conjecture that, in general, the construction mode (see Section 1.1.6) of compose_r(**HX**,**TY**,Y,**Z**) is compose_r(***may***, ***must***, res, ***may***),

where the first ***may*** denotes *may,…,may* with *h* occurrences of *may*, the second ***may*** denotes *may,…,may* with *z* occurrences of *may,* and ***must*** denotes *must,…,must* with *t* occurrences of *must* (remember that *h* is the number of heads HX_i, that *z* is the number of passive parameters, and that *t* is the number of tails TY_i).

Indeed, the TY_i being obtained through recursion, they must all somehow be used to construct Y, because some of the recursive calls would otherwise have been useless. The HX_i need not always be used to construct Y, as it depends on the particular program. So there is no fixed mode for the head(s) of the induction parameter, and their most general mode thus is *may.* The passive parameter(s) **Z** also need not always be used to construct Y. So there also is no fixed mode for the passive parameter(s), and their most general mode thus also is *may.*

Similarly, one can argue that the mode of solve_r(X,Y,**Z**) is solve_r(*may*,*res*,***may***), where ***may*** denotes *may,…,may* with *z* occurrences of *may.* The inductive parameter X and the passive parameter(s) **Z** need not always be used to construct the result parameter Y. So there are no fixed modes for X and **Z**, and their most general mode thus is *may.*

The evidence abduced for the open relations p and q needs to be processed according to the Program Closing Method so that admissible clauses (see Section 1.1.6) for the open relations p and q are obtained. We give an algorithm below for the realization of this process (note that solve_r plays the role of p, and compose_r plays the role of q):

Algorithm 8: induce(pEvidence,qEvidence,pMode,qMode, pClauses,qClauses)

```
Inputs: pEvidence, qEvidence, pMode, qMode
Outputs: pClauses, qClauses
divide the (evidence) clause set for q, i.e. qEvidence, into a minimal number of subsets (called cliques) of which any two elements have an admissible lgθ, i.e. qCliques (see [10] for an efficient algorithm for this NP-complete problem)
qCliques := division(qEvidence,qMode)
analyze every such clique: if the lgθ of the counterpart subset of the clauses for p is also admissible, then delete the clique from the clauses for q; otherwise delete that counterpart subset from the clauses for p, and thus obtain NewqCliques and NewpEvidence.
```

```
    ⟨NewqCliques, NewpEvidence⟩ := prune(qCliques,pMode,pEvidence)
    take the lgθs of the remaining cliques, i.e. NewqCliques, as clauses of q, i.e.
    qClauses
    qClauses := {lgθ(c)|c∈NewqCliques}
    divide the remaining clause set for p, i.e. NewpEvidence, into a minimal number
    of cliques such that any two elements in each clique have an admissible lgθ, i.e.
    pCliques
    pCliques := division(NewpEvidence,pMode)
    build admissible clauses, i.e. pClauses, of the p from their lgθs, i.e. pCliques
    pClauses := {lgθ(c)|c∈pCliques}
```

Let us now turn back to the synthesis of a program for delOdds and see how the "closing" of open relations solve_delOdds and compose_delOdds is done according to Algorithm 8. The evidence collected for the open relations solve_delOdds and compose_delOdds is (see previous sub-section):

solve_delOdds([A,B],[A,B]) ← even(A), even(B)	(1)	compose_delOdds(A,[B],[A,B]) ← even(A)
solve_delOdds([A,B],[B]) ← odd(A), even(B)	(2)	compose_delOdds(A,[B],[B]) ← odd(A)
solve_delOdds([A],[A]) ← even(A)	(3)	compose_delOdds(A,[],[A]) ← even(A)
solve_delOdds([A],[]) ← odd(A)	(4)	compose_delOdds(A,[],[]) ← odd(A)
solve_delOdds([],[]) ←	(5)	(no counterpart)

Following the statements of Algorithm 8, DIALOGS-II first divides the compose_delOdds evidence into the following cliques:

compose_delOdds(A,B,[A|B]) ← even(A) (1,3)
compose_delOdds(A,B,B) ← odd(A) (2,4)

where the first clique is constructed by taking the lgθ of (1) and (3), and the second one by taking the lgθ of (2) and (4) of the compose_delOdds evidence. Next, it analyzes the counterpart sets for solve_delOdds. That is, it takes the lgθ of (1) and (3), as well as the lgθ of (2) and (4) of the solve_delOdds evidence, and thus obtains:

solve_delOdds([A|B],[A|B]) ← even(A), even(C) (1,3)
solve_delOdds([A|B],B) ← odd(A), T (2,4)

None of these two clauses is admissible since the first one contains a literal, i.e. even(C), in its body, which has an unlinked variable, i.e. C. And, the second one is not admissible because the body contains T (see Section 1.1.6). Thus, the counterpart sets of solve_delOdds, i.e. {(1), (3)} and {(2), (4)} are eliminated from the solve_delOdds evidence set and the cliques of compose_delOdds are kept. The lgθs of these two cliques become thus clauses of compose_delOdds, namely the clauses that will be in the final program. The remaining set for solve_delOdds is

```
solve_delOdds([],[]) ←
```

and since this set is a clique and is admissible, its lgθ (i.e. itself) becomes a clause for solve_delOdds.

Now, the open relations solve_delOdds and compose_delOdds are "closed", that is they have an interpretation, and the open program constructed from the initial schema is also "closed" since it has no open relations. The final step in the synthesis is adding the clauses of the open relations to the open program to close the open program. In that way, the final program becomes:

```
delOdds(A,B) ← solve_delOdds(A,B)
delOdds(A,B) ← decompose_delOdds(A,C,D), delOdds(D,E),
                                        compose_delOdds(C,E,B)
decompose_delOdds(F,G,H) ← F=[G|H]
solve_delOdds([],[]) ←
compose_delOdds(A,B,[A|B]) ← even(A)
compose_delOdds(A,B,B) ← odd(A)
```

This program is correct with respect to its specification. Post-synthesis transformations that optimize the final programs are not our concern in this thesis. See [6] if you want to know more about them.

2.5 Evaluation of the Program Closing Method

Finding a program for the open relation of the recursive clause of an open program, i.e. the relation q, via the Program Clausing Method assumes that there is a finite non-recursive program for that relation. However such is not always the case. That is, there might be a recursive one instead. In other words, the system might have to do a necessary predicate invention.

2.5.1 Necessary Predicate Invention

How can the system possibly decide that the result of the Program Closing Method is wrong, that is that the finite non-recursive program that was induced for the relation q via the Program Closing Method is incomplete, and that it has to invent a predicate with a recursive program after rejecting the result of the Program Closing Method? These questions imply that some heuristic needs to be used for detecting and handling necessary predicate invention [12] [25].

Since the Program Clausing Method has been devised to always succeed (indeed, in the worst case, it divides a clause set into cliques of one element each), a heuristic is needed for rejecting the results of the Program Clausing Method and conjecturing necessity of the predicate invention. For the time being, we do not have an acceptable heuristic that frequently correctly conjectures necessary predicate invention, whenever there is a need to synthesize a recursive program. Therefore, in DIALOGS-II, the decision of predicate invention is specifier-dependent. That is, the specifier is asked whether the system should reject the result of the Program Closing Method and synthesize a recursive program (do a necessary predicate invention), or whether it should use the result of the Program Closing Method. If the result of the Program Clausing Method is rejected by the specifier, then DIALOGS-II re-invokes itself under the assumption that a recursive logic program exists for the open relation.

In general, DIALOGS-II is called with a *start program*: this is the empty set in the case of a new synthesis (for the *top-level predicate*), or a set of clauses for a (unique) top-level predicate and its (directly or indirectly) used predicates, in case DIALOGS-II is used (possibly by itself) for a necessary invention of a predicate that is (directly or indirectly) used by the top-level predicate. In case there is a predicate invention, the new program synthesized for the new predicate is added to the start program, otherwise the clauses induced by the Program Closing Method are added to the start program, yielding the final program.

We saw how query generation and answering take place when there is no predicate invention and how the result of the Program Closing Method is used for “closing” the open relations during the synthesis of a program for delOdds. Now let us see how this is done in case of necessary predicate invention: when a necessity of predicate invention is conjectured, query generation during the synthesis of the new predicate is always done for the top-level predicate, but resolution will eventually be blocked by an open relation of the current predicate and thus the system will extract a question for it in terms of the top-level one. This is because the user does not always (see the next sub-section for an exceptional case) need to know the predicate being invented, but s/he has to know the top-level predicate since otherwise s/he would not even have the need for a program for the top-level predicate. Thus, DIALOGS-II generates queries for the new predicate in terms of the top level predicate, but resolution is eventually blocked by an open atom of the program of the new predicate, i.e. current predicate, and extract a question for it in terms of the top-level one.

Now, we introduce two new concepts: the concept of giving hints and the concept of calling DIALOGS-II in a certain mode: *aloud* or *mute*. Let us first discuss the concept of giving hints: hints about the roles of the parameters of a certain parameter declaration can be given to the system. In a recursive call of DIALOGS-II itself, it is possible to hint about the parameter roles of the new predicate (how this is done will be explained later). So, we can say that DIALOGS-II can be called with hints about the roles of the parameters (if there are any hints), where the initial call of DIALOGS-II for the top-level predicate is done with an empty hint list. DIALOGS-II has preference of hints over defaults. In other words, if there are any hints, then the system uses these hints instead of using the defaults.

Now let us introduce the concept of calling the system in *mute* or *aloud* mode: DIALOGS-II is said to be in *aloud* mode when it asks the specifier for a predicate declaration, a schema, a strategy, parameter roles and a decomposition operator, and gets the answer from the specifier whereas it is said to be in *mute* mode when the specifier is queried for nothing, where the system itself answers the questions by itself. By default, the system is in *aloud* mode when it starts synthesis, but it is called in *mute* mode when there is necessary predicate invention. Now, we give an algorithm that realizes all the observations and discussions explained so far. What this algorithm basically does is that it *evaluates* the result of the Program Closing Method based on the specifier's evaluation of the Program Closing Method and calls DIALOGS-II recursively, in *mute* mode, to synthesize a recursive program for the predicate `q` if predicate invention is necessary, otherwise it uses the result of the Program Closing Method to produce a non-recursive program for the relation `q`:

Algorithm 9: evaluate(Schema,Strategy,CurrOpenPgm, pClauses,qClauses,PredDecl,ParamRoles,Pgm)

```
Inputs: Schema, Strategy, CurrOpenPgm, pClauses, qClauses,
PredDecl, ParamRoles
Outputs: Pgm
display the result of the Program Closing Method
display(pClauses,qClauses)
ask the specifier if predicate invention is necessary
Answer := ask('Please evaluate the Program Closing Method: need for
recursive synthesis? [yes/no]')
  if Answer=yes then
    determine the predicate declaration for the new predicate for which a
    recursive program is being synthesized using ParamRoles of TopPred,
    where TopPred is the name of the predicate given in PredDecl
    NewPredDecl := predDecL(ParamRoles)
    TopPred := predName(PredDecl)
    add the clauses for the relation p, i.e. SelectedpClauses, which are from
    pClauses and have no counterparts among the clauses of qClauses, to
    CurrOpenPgm to obtain NewOpenPgm
    SelectedpClauses := select(pClauses,qClauses)
    NewOpenPgm := CurrOpenPgm ∪ SelectedpClauses
    construct hints about the roles of the parameters
    Hints := constructHints(NewPredDecl)
    setMode(mute)
    call DIALOGS-II recursively with the new predicate declaration and hints to
    induce a program for the new predicate
    dialogsII(Schema,Strategy,NewPredDecl,NewOpenPgm,Hints,
    TopPred,Pgm)
  else
    add the clauses pClauses and qClauses to CurrOpenPgm to obtain Pgm
    Pgm := CurrOpenPgm ∪ pClauses ∪ qClauses
```

Note that in Algorithm 9, DIALOGS-II is now called with NewPredDecl and Hints about the parameter roles, where the final program for the new predicate will be added to NewOpenPgm, which is an open program (whose relation q is still open) for TopPred.

Synthesis of a Program for reverse(L,R). Now we will illustrate how Algorithm 9 works by means of the synthesis of a program for reverse(L,R), where reverse(L,R) holds iff list R is the reverse of list L. Since we already discussed the first two statements of the basic synthesis algorithm, i.e. Algorithm 1, and first two statements of Algorithm 2, in terms of the synthesis of a program for delOdds, we will skip these statements in the illustration of the synthesis of a program for reverse(L,R), where we will only give the results of these statements.

By the execution of the first statement (execution of the strategy) of Algorithm 2, the following open program for reverse(L,R) has been generated:

```
reverse(A,B) ← solve_reverse(A,B)
reverse(A,B) ← decompose_reverse(A,C,D), reverse(D,E),
                                                compose_reverse(C,E,B)
decompose_reverse(F,G,H) ← F=[G|H]
```

where A is the induction parameter and B is the result parameter.

Remember that by executing the second statement of Algorithm 2, the evidence for the open relations, i.e. p and q, is abduced. So, at the end of the second statement, the evidence for solve_reverse and compose_reverse is as given below in the form of counterparts:

solve_reverse([A],[A]) ←	(1)	compose_reverse(A,[],[A]) ←
solve_reverse([A,B],[B,A]) ←	(2)	compose_reverse(A,[B],[B,A]) ←
solve_reverse([A,B,C],[C,B,A]) ←	(3)	compose_reverse(A,[B,C],[B,C,A]) ←
solve_reverse([A,B,C,D],[D,C,B,A]) ←	(4)	compose_reverse(A,[B,C,D],[B,C,D,A]) ←
solve_reverse([],[]) ←	(5)	(no counterpart)

where the Program Closing Method results in the following clauses for the open relation solve_reverse (note that there is no compose_reverse clause):

```
solve_reverse(A,A) ←
solve_reverse([A,B],[B,A]) ←
solve_reverse([A,B,C],[C,B,A]) ←
```

Now, it is time to query the specifier about the result of the Program Closing Method to conjecture whether predicate invention is necessary or not.

```
Please evaluate the Program Closing Method: need for
recursive synthesis? [yes/no] yes
```

The specifier here answers the query by *yes* believing that there exists a recursive program for the predicate of the recursive clause of the open program, i.e. compose_reverse, rejecting the result of the Program Closing Method. Since the system now knows that it should synthesize a recursive program for the compose_reverse predicate, it needs to call itself recursively. But, before doing this it should first elaborate a predicate declaration for the predicate, and construct hints about the parameter roles, and compute the new start program for the new synthesis by adding the clauses for the relation solve_reverse that have no counterparts among the clauses of compose_reverse.

Determination of a Predicate Declaration for the New Predicate. Now, let us go through the steps of determination of a predicate declaration for the new predicate one by one to see how they are realized. First, we discuss how the new predicate declaration is elaborated. A predicate declaration has two components: the name of the predicate and the list of parameters together with their types. The name of the new predicate is already known, which is compose_reverse. The list of parameters together with their types is elaborated as follows: it is known that the new predicate has three parameters. The type of the first parameter is found to be int, since in the open program given above the first parameter of compose_reverse, i.e. C, is the head (namely, an element) of the list A, where the type of the parameter A is list(int). The type of the second parameter is found to be list(int), since in the open program the second parameter of compose_reverse, i.e. E, is the result parameter of the recursive call, i.e. reverse(D,E), where the result parameter of the reverse predicate is of type list(int). Finally, the third parameter is found to be of type list(int), since it also is the result parameter of the open program, where its type is list(int). Thus, using the information about the name of the new predicate and the parameters together with their types, the predicate declaration for the new predicate is constructed as shown below:

```
compose_reverse(HL:int,TR:list(int),R:list(int))
```

Now, the system has a predicate declaration of the new predicate for which it will call itself to induce a program.

Construction of Hints. Next, it has to construct hints about the parameter roles, i.e. which parameter is the induction parameter, which one is (are) the result parameter(s) (if any), and which one is (are) the passive parameter(s) (if any), in order to call itself in *mute* mode with these hints (remember that DIALOGS-II has a preference of hints over defaults in *mute* mode). It is reasonable that R (see the predicate declaration above) is hinted as the result parameter since the corresponding parameter B in the open program (see the open program on page 44) is the result parameter of the program, and it is also reasonable to hint TR as the induction parameter since it is of an inductively defined type, and finally to hint the remaining parameter HL as the passive parameter. In general, the result parameter of the open relation q in the open program can be hinted as a result parameter for the new predicate, a parameter which is the result parameter of the recursive call in the open program can be hinted as an induction parameter if it is of an inductively defined type, and the remaining parameters as the passive parameters. Here we described the determination of hints about the parameter roles for a divide-and-conquer schema, since we are illustrating the synthesis of a program that fits a divide-and-conquer schema. The construction of hints would be different if the schema were another one, e.g. descending-generalization, since the parameter roles of the schema would be different.

Construction of a Start Program for the New Synthesis. What DIALOGS-II does after elaboration of the new predicate declaration and construction of hints is that it constructs a start program for the new synthesis by using the evidence clauses abduced during execution of the second statement of Algorithm 2. How this is done is as follows: the system adds the abduced clauses for the relation p that have no counterparts among the abduced clauses for the relation q to the open program to obtain the start program for the new synthesis. The clause

```
solve_reverse([],[]) ←
```

has no counterparts among the abduced clauses for the relation compose_reverse (see page 45). Thus the start program for the new synthesis is:

```
reverse(A,B) ← solve_reverse(A,B)
reverse(A,B) ← decompose_reverse(A,C,D), reverse(D,E),
                                    compose_reverse(C,E,B)
```

```
decompose_reverse(F,G,H) ← F=[G|H]
solve_reverse([],[]) ←
```

Now, it is time for the system to re-invoke itself on this start program using the new predicate declaration and the hints.

Calling DIALOGS-II Recursively. Before calling the system recursively, the synthesis mode is converted into *mute* mode. DIALOGS-II first determines the roles of the parameters that are given inside the predicate declaration using the hints and the decomposition operator using the defaults. Next, an open program is generated for the new predicate, and this open program is added to the start program to obtain the new open program that will be used for the abduction of the new evidence for the open relations of the open program of the new predicate. The second and third statements (abduction of evidence and induction of clauses) of Algorithm 2 are then executed using this new open program to "close" the open relations of the open program of the new predicate. Let us now see how all this is done during the synthesis of a program for the reverse(L,R) predicate.

DIALOGS-II first determines the roles of the parameters of the predicate declaration:

```
compose_reverse(HL:int,TR:list(int),R:list(int))
```

using the hints determined previously. That is, the induction parameter is TR, the result parameter is R, and the passive parameter is HL (note that DIALOGS-II does not query the user for that since it uses the hints).

```
Induction parameter? {TR} TR
Result parameter? {R} R
Passive parameter(s)? {[HL]} [HL]
```

Next, it determines decompose_compose_reverse by using the default one.

```
Decomposition operator?
{decompose_compose_reverse(L,H,T) ← L =[H|T]}
decompose_compose_reverse(L,H,T) ← L=[H|T]
```

Next, it generates the following open program using decompose_compose_reverse:

```
reverse(A,B) ← solve_reverse(A,B)
reverse(A,B) ← decompose_reverse(A,C,D), reverse(D,E),
```

compose_reverse(C,E,B)

decompose_reverse(F,G,H) ← F=[G|H]

solve_reverse([],[]) ←

compose_reverse(G,M,N) ← solve_compose_reverse(G,M,N)

compose_reverse(G,H,I) ← decompose_compose_reverse(H,J,K),
    compose_reverse(G,K,L), compose_compose_reverse(J,L,I,G)

decompose_compose_reverse(F,G,H) ← F=[G|H]

Now, it is time to abduce evidence for the open relations, i.e. compose_compose_reverse and solve_compose_reverse, where during the abduction of the evidence for these relations, the system does not query the specifier, but uses the shortcuts for the top-level predicate reverse, except in the case where there is no shortcut left after using the available shortcuts:

reverse([],[]) ← (s1)
reverse([A],[A]) ← (s2)
reverse([A,B],[B,A]) ← (s3)
reverse([A,B,C],[C,B,A]) ← (s4)
reverse([A,B,C,D],[D,C,B,A]) ← (s5)

After that point, the SLD resolution of a goal for the top-level predicate is blocked by an open atom, and the system extracts a query for this open atom, where the answer to that query is found using the shortcuts of the top-level predicate. Let us now go through the steps of "closing" the open relations of the open program given above.

The most general form of the goal when the size of the induction parameter, i.e. A, is 0 is the following: ← reverse([ ],X). This goal is first tried to be resolved with the recursive clause of the reverse predicate, where this attempt fails since resolving decompose_reverse when the induction parameter A is the empty list, i.e. [], is impossible. The system next resolves the goal with the non-recursive clause of the open program. As a result of this resolution, the goal

← true

is reached because the predicate solve_reverse is already closed (there are clauses for the solve_reverse predicate), and thus there is no need to abduce evidence for it.

Next, the goal ← reverse([X],Y), where the induction parameter is a one-element list, is resolved with the clauses of the reverse predicate, first with the recursive clause yielding the goal:

← decompose_reverse([X],C,D), reverse(D,E),
compose_reverse(C,E,Y)

Resolving decompose_reverse([X],C,D) and the resulting equality atom, and using the shortcut s1 gives:

← compose_reverse(X,[],Y)

Since there is no shortcut for compose_reverse(X,[ ],Y) (shortcuts obtained before starting the new synthesis are not kept, to prevent them from being accidentally used by the new synthesis as shortcuts, see Algorithm 5), the goal ← compose_reverse(X,[],Y) is resolved with the non-recursive clause of compose_reverse (note that the recursive clause cannot be resolved since the induction parameter, i.e. [], cannot be decomposed) yielding the goal:

← solve_compose_reverse(X,[],Y)

There is neither a shortcut nor a program for the predicate solve_compose_reverse, so resolving this goal is impossible. Therefore, it is time to make a query out of this goal. Since there is a shortcut, i.e. s2, the system uses the shortcut s2 to abduce the following evidence and shortcuts for solve_compose_reverse and compose_reverse without any need for a query:

solve_compose_reverse(X,[],[X]) ←
compose_reverse(X,[],[X]) ← (s6)

The system resolves the goal ← reverse([X],Y) with the non-recursive clause of the reverse predicate. As a result of this resolution, the goal

← true

is reached because the predicate solve_reverse is already closed (there are clauses for the solve_reverse predicate), and thus there is no need to abduce evidence for it.

Next, the goal ← reverse([X,Y],W) is resolved with the recursive clause of the reverse predicate, yielding the goal:

← decompose_reverse([X,Y],C,D), reverse(D,E),
compose_reverse(C,E,W)

Resolving decompose_reverse([X,Y],C,D) and the resulting equality atom, and using the shortcut (s2), the goal becomes

← compose_reverse(X,[Y],W)

Since there is no shortcut for compose_reverse(X,[Y],W), it is resolved with the recursive clause of the compose_reverse predicate yielding the goal

← decompose_compose_reverse([Y],J,K), compose_reverse(X,K,L),
compose_compose_reverse(J,L,W,X)

Resolving decompose_compose_reverse([Y],J,K) and the resulting equality atom, and using (s6) gives

← compose_compose_reverse(Y,L,W,X)

Resolving this goal is impossible since there is neither a shortcut nor a clause for compose_compose_reverse. So, the following evidence and shortcuts are abduced using the shortcut s3:

compose_compose_reverse(Y,[X],[Y,X],X) ←
compose_reverse(X,[Y],[Y,X) ← (s7)

Upon backtracking, the goal

← compose_reverse(X,[Y],W)

is resolved with the non-recursive clause of the compose_reverse yielding the goal:

← solve_compose_reverse(X,[Y],W)

Using the shortcut s7 the following evidence is abduced:

solve_compose_reverse(X,[Y],[Y,X]) ←

Next, this resolution process is also done for the most general values of the induction parameter A of the reverse predicate (see the open program for reverse) when the size of the induction parameter is three and then four, i.e. reverse([X,Y,W],V), and reverse([X,Y,W,V],Z). And, as a result of this process the following evidence and shortcuts are abduced:

solve_compose_reverse(X,[W,Y],[W,Y,X]) ← (2)
solve_compose_reverse(X,[V,W,Y],[V,W,Y,X]) ← (1)
compose_compose_reverse(X,[W,Y],[X,W,Y],Y) ← (2)
compose_compose_reverse(X,[W,Y,V],[X,W,Y,V],V) ← (1)

```
compose_reverse(X,[W,Y],[W,Y,X]) ←
compose_reverse(X,[V,W,Y],[V,W,Y,X]) ←
```

The evidence given above for solve_compose_reverse and compose_compose_reverse together with the following evidence

```
solve_compose_reverse(X,[],[X]) ←                  (no counterpart)
solve_compose_reverse(X,[Y],[Y,X]) ←                            (3)
compose_compose_reverse(Y,[X],[Y,X],X) ←                        (3)
compose_reverse(X,[],Y) ←
compose_reverse(X,[Y],[Y,X]) ←
```

abduced previously is input to the Program Closing Method in order to find programs for these open relations. Following Algorithm 8, DIALOGS-II first divides the compose_compose_reverse evidence into a clique and computes its lgθ:

```
compose_compose_reverse(L,[M|N],[L,M|N],P) ←               (1,2,3)
```

where the clique is constructed by taking the lgθ of (1), (2) and (3) of compose_compose_reverse evidence (see Algorithm 8). Next, it analyzes the counterpart set for solve_compose_reverse. That is, it takes the lgθ of (1), (2), and (3) of the solve_compose_reverse evidence, and thus obtains:

```
solve_compose_reverse(A,[B|C],[B,D|E]) ←                   (1,2,3)
```

Since this clause is not admissible (see Section 1.1.6), it is not kept. The remaining clause

```
solve_compose_reverse(X,[],[X]) ←
```

that has no counterpart is kept in the final program. Thus, as a result of the Program Closing Method the following two clauses are induced

```
solve_compose_reverse(K,[],[K]) ←
compose_compose_reverse(L,[M|N],[L,M|N],P) ←
```

Adding these clauses to the open program gives the following program for the reverse predicate, which is correct with respect to its specification:

```
reverse(A,B) ← solve_reverse(A,B)
reverse(A,B) ← decompose_reverse(A,C,D), reverse(D,E),
                                        compose_reverse(C,E,B)
decompose_reverse(F,G,H) ← F=[G|H]
```

```
solve_reverse([],[]) ←
compose_reverse(P,V,W) ← solve_compose_reverse(P,V,W)
compose_reverse(P,Q,R) ← decompose_compose_reverse(Q,S,T),
   compose_reverse(P,T,U), compose_compose_reverse(S,U,R,P)
decompose_compose_reverse(F,G,H) ← F=[G|H]
solve_compose_reverse(K,[],[K]) ←
compose_compose_reverse(L,[M|N],[L,M|N],P) ←
```

2.5.2 Handling the Sparseness Problem

DIALOGS-II faces the *sparseness* problem [19] when not every value of the induction parameter of the new predicate, i.e. q, is "reachable" by the values of the induction parameter of the top-level predicate. That is, queries about the new predicate cannot always be asked in terms of the top-level one. To show how we solve this problem, we will examine the synthesis of a program for the factorial predicate, where factorial(N,F) holds iff natural number F is the factorial of natural number N. What happens during the synthesis of a factorial program, in short, is that the synthesis requires the invention of the multiplication predicate, where multiplication(A,B,C) holds iff natural number C is the product of natural numbers A and B, but actually only uses a sparse subset of the multiplication relation. That is, it uses the following subset of the multiplication relation.

multiplication(s(0),s(0),s(0))
multiplication(s(0),s^2(0),s^2(0))
multiplication(s^2(0),s^3(0),s^6(0))
multiplication(s^6(0),s^4(0),s^{24}(0))
...

So, the evidence abduced for the open relations of the open program of the multiplication relation is a sparse set of evidence from which it is not possible to induce a correct and complete multiplication program, nor in turn a correct and complete factorial program with respect to its specification. Here, we introduce a new solution to the sparseness problem. Before explaining this new approach, let us first give a new conjecture.

The "Yılmaz Conjecture". We conjecture that if there is a relation such that during the synthesis of a program for that relation the sparseness problem occurs, then the specifier should be able to answer the queries related to the relations that are intrinsic to the relation being induced (this is the exception that was mentioned in Section 2.5).

For instance, during the synthesis of a program for factorial, if the specifier is able to answer the query

`When does factorial(`s^3`(0),L) hold?`

then s/he should also be able to answer the following query about multiplication, after having seen some evidence of the multiplication relation that was abduced and is different from the one given below:

`When does multiplication(`s^2`(0),`s^3`(0),M) hold?`

since, what s/he is actually doing while finding an answer to the query of the factorial relation is that s/he is using the multiplication relation, because otherwise s/he would not be able to answer the query about the factorial relation. In other words, multiplication is "intrinsic" to factorial.

In our approach to handling the sparseness problem, we use the idea given by the conjecture above. Before explaining how we use this idea, let us first investigate how the system conjectures that there is a sparseness problem. In DIALOGS-II, this detection is done by means of a heuristic. How this heuristic works is as follows: if the abduced evidence for the open relations in the open program for the new predicate (the evidence for the solve_compose_q and compose_compose_q) is *unbalanced*, that is, if there are at least three more solve_compose_q clauses than compose_compose_q clauses, then the system conjectures that there is a sparseness problem. The number three has been determined empirically (e.g. based on the results obtained during the synthesis of a program for the factorial predicate). When the system conjectures that there is a sparseness problem, the evidence abduced for solve_compose_q and compose_compose_q is discarded, and a new synthesis, in *aloud* mode, is started for the q predicate, after letting the specifier know that there will be a new synthesis for the new predicate, and s/he would need to answer the queries of that new synthesis. Let us now refine the algorithm *evaluate* (Algorithm 9) such that it conjectures the sparseness problem:

Algorithm 10: evaluate(Schema,Strategy,CurrOpenPgm, pClauses,qClauses,PredDecl,ParamRoles,Pgm)

```
Inputs: CurrOpenPgm, pClauses,qClauses,TopPred,ParamRoles
Outputs: Pgm
display the result of the Program Closing Method
display(pClauses,qClauses)
ask the specifier if predicate invention is necessary
Answer := ask('Please evaluate the Program Closing Method: need for
recursive synthesis? [yes/no]')
  if Answer = yes then
    determine the predicate declaration for the new predicate for which a
    recursive program is being synthesized using ParamRoles of TopPred,
    where TopPred is the name of the predicate given in PredDecl
    NewPredDecl := predDecL(ParamRoles)
    TopPred := predName(PredDecl)
    add the clauses for the relation p, i.e. SelectedpClauses, which are from
    pClauses and have no counterparts among the clauses of qClauses, to
    CurrOpenPgm to obtain NewOpenPgm
    SelectedpClauses := select(pClauses,qClauses)
    NewOpenPgm := CurrOpenPgm ∪ SelectedpClauses
    check if there is sparseness problem by calling DIALOGS-II recursively with
    the new predicate declaration in mute mode using the heuristic
    qAndpEvidence := collectAssertedEvidence(q,p)
    SynthesisMode := getMode()
    setMode(mute)
    dialogsII(Schema,Strategy,NewPredDecl,NewOpenPgm,[],
    TopPred, Pgm)
    setMode(SynthesisMode)
    Sparseness := sparsenessHeuristic(qAndpEvidence)
    if Sparseness = no then
      Hints := constructHints(NewPredDecl)
      setMode(mute)
      call DIALOGS-II recursively with the new predicate declaration and
```

```
            hints to induce a program for the new predicate such that final Pgm is obtained
            dialogsII(Schema,Strategy,NewPredDecl,NewOpenPgm,Hints,TopPred,Pgm)
        else
            let the specifier know that a new synthesis for new predicate is being started and display the abduced clauses for the new predicate
            NewPred := predName(NewPredDecl)
            Clauses := collectAssertedEvidence(NewPred)
            display(Clauses)
            setMode(aloud)
            Hints := constructHints(NewPredDecl)
            call DIALOGS-II recursively with the new predicate declaration and an empty hint list to induce a program for the new predicate in aloud mode
            dialogsII(Schema,Strategy,NewPredDecl,NewOpenPgm,Hints,NewPred,Pgm)
    else
        add the clauses pClauses and qClauses to the CurrOpenPgm to obtain Pgm
        Pgm := CurrOpenPgm ∪ pClauses ∪ qClauses
```

Note that if there is a sparseness problem, then the system will call DIALOGS-II recursively to induce a new program for the new predicate.

Now, let us examine the synthesis of a program for the factorial predicate. Suppose that the following open program for the factorial predicate is generated at the end of the execution of the first statement of Algorithm 2:

```
factorial(A,B) ← solve_factorial(A,B)
factorial(A,B) ← decompose_factorial(A,C,D), factorial(C,E),
compose_factorial(D,E,B)
decompose_factorial(F,G,H) ← F=s(G), H=F
```

And, also suppose that the Program Closing Method yields the following clauses for solve_factorial (note that no clause for compose_factorial has been induced):

solve_factorial(s^3(0),s^6(0)) ←
solve_factorial(s(A),s(A)) ←
solve_factorial(0,s(0)) ←

Now, suppose that the specifier is asked to evaluate the result of the Program Closing Method, and s/he rejects it (s/he thinks that predicate invention is necessary), and thus the open program given above becomes (see Algorithm 10):

factorial(A,B) ← solve_factorial(A,B)
factorial(A,B) ← decompose_factorial(A,C,D), factorial(C,E),
compose_factorial(D,E,B)
decompose_factorial(F,G,H) ← F=s(G), H=F
solve_factorial(0,s(0)) ←

Next, the predicate declaration for compose_factorial is determined, as it was done for the compose_reverse predicate, which is:

compose_factorial(A:nat,B:nat,C:nat)

Now, it is time for the system to detect if there is a sparseness problem. The sparseness problem is detected by calling DIALOGS-II in *mute* mode using the new predicate declaration (the shortcuts abduced for the factorial predicate previously are used for this new synthesis). Thus, the system abduces the following evidence for solve_compose_factorial (note that no clause for compose_compose_factorial is induced) at the end of the Program Closing Method of this new synthesis:

solve_compose_factorial(s(0),s(0),s(0)) ←
solve_compose_factorial(s^2(0),s(0),s^2(0)) ←
solve_compose_factorial(s^3(0),s^2(0),s^6(0)) ←

The system now uses the heuristic to see if there is any sparseness problem: the number of clauses for solve_compose_factorial is three (at least three) more than the number of clauses for compose_compose_factorial. So, a correct program for the relations solve_compose_factorial and compose_compose_factorial, and thus for compose_factorial in turn, cannot be induced from this evidence, and therefore the evidence is eliminated. Thus, a new synthesis for a program for compose_factorial is started in *aloud* mode by letting the specifier know about this:

```
You must know the relation compose_factorial since it
is intrinsic to the factorial relation. The clauses of
```

this relation obtained during the synthesis are given below. The system is starting a new synthesis for that relation, so please answer the queries about it:
compose_factorial(s(0),s(0),s(0)) ←
compose_factorial(s^2(0),s(0),s^2(0)) ←
compose_factorial(s^3(0),s^2(0),s^6(0)) ←

Note that the relation compose_factorial is actually the multiplication relation. And, with the new predicate declaration, the system is called recursively with an empty hint list yielding the resulting open program for compose_factorial at the end of the execution of these statements:

compose_factorial(T,U,V) ← solve_compose_factorial(T,U,V)
compose_factorial(T,U,V) ← decompose_compose_factorial(T,W,X),
    compose_factorial(W,U,Y), compose_compose_factorial(X,Y,V,U)
decompose_compose_factorial(F,G,H) ← F=s(G), H=F

Next, the query session for the synthesis of programs for solve_compose_factorial and compose_compose_factorial takes place to abduce evidence for these relations, where the specifier answers the queries:

When does compose_factorial(0,A,B) hold? B=0.
When does compose_factorial(s(0),A,B) hold? B=A.
When does compose_factorial(s(s(0)),A,B) hold? B=A+A.
When does compose_factorial(s(s(s(0))),A,B) hold?
B=A+A+A.

The abduced evidence from this query is:

solve_compose_factorial(0,A,0) ←
solve_compose_factorial(s(0),A,A) ←
solve_compose_factorial(s(s(0)),A,A+A) ←
solve_compose_factorial(s(s(s(0))),A,A+A+A) ←
compose_compose_factorial(s(0),0,A,A) ←
compose_compose_factorial(s(s(0)),A,A+A,A) ←
compose_compose_factorial(s(s(s(0))),A+A,A+A+A,A) ←

From this evidence, using the Program Closing Method, the following program is induced for compose_factorial:

```
compose_factorial(T,U,V) ← solve_compose_factorial(T,U,V)
compose_factorial(T,U,V) ← decompose_compose_factorial(T,W,X),
    compose_factorial(W,U,Y), compose_compose_factorial(X,Y,V,U)
decompose_compose_factorial(F,G,H) ← F=s(G), H=F
solve_compose_factorial(0,S,0) ←
solve_compose_factorial(s(0),R,R) ←
compose_compose_factorial(s(s(O)),P,P+Q,Q) ←
```

Finally, this new program for compose_factorial is added to the open program for factorial yielding the following program for the factorial predicate:

```
factorial(A,B) ← solve_factorial(A,B)
factorial(A,B) ← decompose_factorial(A,C,D), factorial(C,E),
                                                     compose_factorial(D,E,B)
decompose_factorial(F,G,H) ← F=s(G), H=F
solve_factorial(0,s(0)) ←
compose_factorial(T,U,V) ← solve_compose_factorial(T,U,V)
compose_factorial(T,U,V) ← decompose_compose_factorial(T,W,X),
   compose_factorial(W,U,Y), compose_compose_factorial(X,Y,V,U)
decompose_compose_factorial(F,G,H) ← F=s(G), H=F
solve_compose_factorial(0,A,0) ←
solve_compose_factorial(s(0),B,B) ←
compose_compose_factorial(s(s(C)),D,D+E,E) ←
```

where this factorial program is correct with respect to its specification. If we partially evaluate this program, then we obtain the following program that is more “readable”:

```
factorial(0,s(0)) ←
factorial(A,B) ← A=s(C), factorial(C,E), compose_factorial(A,E,B)
compose_factorial(0,A,0) ←
compose_factorial(s(0),B,B) ←
compose_factorial(T,U,V) ← T=s(W), compose_factorial(W,U,Y),
                                     compose_compose_factorial(T,Y,V,U)
compose_compose_factorial(s(s(C)),D,D+E,E) ←
```

Chapter 3

Comparison of DIALOGS-II with other ILP Systems

We compare DIALOGS-II with other ILP systems in terms of the evidence given as input to the system, and in terms of the power of their schemata. We first discuss (in Section 3.1) the evidence given in the form of examples and given in the form of syntactic bias (see Section 1.1.3), and then (in Section 3.2) we compare other ILP systems with DIALOGS-II in terms of the schemata available to these systems.

3.1 Comparison in Terms of the Evidence

FOIL [24] is a general purpose system that induces recursive and non-recursive logic programs. In order to learn a recursive program for length(A,L), where length(A,L) holds iff natural number L is the length of the list A, it needs in the order of thousands of positive and negative examples. On the other hand, DIALOGS-II can synthesize a recursive logic program for length(A,L) from as few as three positive examples. The reason for FOIL to consume that many examples for the synthesis of such a simple recursive program is that it is a general purpose synthesizer that does not differentiate between the synthesis of non-recursive programs and the synthesis of recursive ones. This leads to poor “recursion” handling, and, as a result, the necessity of thousands of examples for “encoding” the recursion. As advocated by Biermann [4], we believe that it is more efficient to try a suite of fast and reliable class-specific synthesizers (and, if necessary, to fall back onto a general purpose synthesizer) than to simply run such a slow, if not unreliable, general-purpose synthesizer.

The Tracy system [3] gets a description of the hypothesis space in the form of a syntactic bias and induces recursive logic programs using that bias. Suppose that for the `append` predicate (where `append(A,B,C)` holds iff list `C` is the concatenation of list `A` in front of list `B`), the following bias, positive and negative examples, and mode declaration are given as inputs, where the program and mode declaration of the = predicate are considered given as background knowledge:

```
append(A,B,C) ← {B=C, A=[]}
append(A,B,C) ← {A=[H|T], B=[E|F], append(T,{E,B,A},{D,F}), C=[H|D]}
+append([a],[b],[a,b])
–append([a],[b],[a])
–append([a],[b],[b])
append_inout(in,in,out)
```

The curly braces used for writing the body atoms and the parameters denote one element of the powerset of the elements inside the braces. After generating all possible clauses in the hypothesis space encoded by the bias above, the set of clauses used in the derivation of the positive example such that these clauses do not cover any of the two negative examples yields the final program:

```
append(A,B,C) ← B=C, A=[]
append(A,B,C) ← A=[H|T], append(T,B,D), C=[H|D]
```

Note that the recursive call is already encoded in the bias: the technique itself cannot discover recursion. In that sense, the source already knows how to write a possible program for `append`. If the same synthesis would be done with Dialogs-II, the source would not need to know how to write a program for `append`. In fact, this is the ideal scenario since the very aim of a synthesizer is to synthesize a program that is unknown (or not completely known) to the source; it is not to extract a possible program from the evidence that encodes this program.

In summary, Dialogs-II synthesizes recursive logic programs from little evidence, and the source can use Dialogs-II to synthesize a recursive logic program that is unknown to it.

3.2 Comparison in Terms of Schemata

METAINDUCE [18] is almost exactly a subset of DIALOGS-II. Its schema is a particular case of the divide-and-conquer schemata of DIALOGS-II, namely for ternary relations, induction parameter of type list, exactly one base clause (when the list is empty), exactly one recursive clause (when the list is non-empty), and head-tail decomposition of the list (i.e. exactly one recursive call). In other words, the divide-and-conquer schemata that can be used by DIALOGS-II is more powerful: the induction parameter is not necessarily of type list, as it can be of any type that is inductively defined, multiple base clauses and multiple recursive clauses are possible, and the decomposition is not necessarily a head-tail one.

CRUSTACEAN [1] [2] synthesizes recursive logic programs of the following schema:

$\mathsf{p}(A_1,\ldots,A_n) \leftarrow$
$\mathsf{p}(A_1,\ldots,A_n) \leftarrow \mathsf{p}(B_1,\ldots,B_n)$

where the A_i and B_i are terms. This is a very restricted schema compared to the possible divide-and-conquer schemata of DIALOGS-II. It has only one base clause and one recursive clause. Moreover, because of the schema, there is no possibility of any kind of predicate invention.

The schema of the CILP system [19] is a superior to that of CRUSTACEAN:

p(…) ←
p(…) ← p(…)

or, in the case of necessary predicate invention, it is:

q(…) ←
q(…) ← q(…), newp(…)
newp(…) ←
newp(…) ← newp(…)

The CILP schema is superior to the schema of CRUSTACEAN. When there is no predicate invention, the schema of CILP is the same as that of CRUSTACEAN; when there is predicate invention, the schema has one base clause and one recursive clause, which has an invented predicate whose program has only one base clause and one recursive clause. When there is predicate invention, DIALOGS-II invents predicates whose programs are also be of the divide-and-conquer schemata of DIALOGS-II, which implies DIALOGS-II can make use of divide-and-conquer schemata that are more general than that of CILP.

The hypothesis language of the FORCE2 system [8] is two-clause linear and closed recursive *ij*-determinate logic programs. A clause is linear and closed recursive if the body of the clause has a single recursive atom that is closed, i.e. has no output variables. Thus, the schema is:

$$\begin{array}{l} \mathsf{p}(\ldots) \leftarrow \mathsf{q}_1(\ldots), \ldots, \mathsf{q}_m(\ldots) \\ \mathsf{p}(\ldots) \leftarrow \mathsf{r}_1(\ldots), \ldots, \mathsf{r}_n(\ldots), \mathsf{p}(\ldots) \end{array}$$

where each q_k and r_k is an *ij*-determinate literal that is defined in the background knowledge, and the recursive atom $\mathsf{p}(...)$ has no output variables. This schema is restricted with respect to the possible divide-and-conquer schemata of DIALOGS-II since it has only one base clause and only one recursive clause, where the recursive clause has only one recursive call. Moreover, the schema above is further restricted by *ij*-determinacy, where the divide-and-conquer schemata of DIALOGS-II have no such constraint.

In summary, there exist divide-and-conquer schemata that can be used by DIALOGS-II, which are superior to those of all other ILP systems known to us.

3.3 Comparison of DIALOGS-II with DIALOGS

DIALOGS-II enables the specifier to select a certain schema together with a strategy, whereas DIALOGS does not have such a concept of selection of a schema and a strategy, i.e. the concept of schema-guidedness; however, DIALOGS is schema-based (has a hard-wired divide-and-conquer schema together with a strategy), and it was thus the first step towards the schema-guidedness of DIALOGS-II, and, to the best of our knowledge, DIALOGS-II is the first in schema-guided synthesis in the field of ILP.

DIALOGS-II uses the open program approach (a first-order approach) in representing schemata, whereas DIALOGS uses a second-order approach in representing its divide-and-conquer schema. Using the open program approach simplifies the representation and manipulation of the schemata of the system.

DIALOGS-II handles the sparseness problem, thus enabling the system to induce programs, e.g. `factorial`, that were not inducable by DIALOGS.

Another difference between DIALOGS and DIALOGS-II is that the DIALOGS implementation did not make a difference between the semantics of the answers *false* and *stop-it* to the queries. Actually, *false* means that there does not exist any condition such that the goal in the query might hold, whereas *stop-it* means that the specifier wants to stop the query session. In the DIALOGS-II implementation, *false* and *stop-it* have their intended meanings.

DIALOGS-II Uses Clause lgθ. DIALOGS uses term lgθs in its MSG Method, whereas DIALOGS-II uses clause lgθs in its Program Closing Method, since clause lgθ is a more powerful way of handling generality among clauses.

If we had used term lgθ instead of clause lgθ in the Program Closing Method, then the order of the atoms inside a clause would matter. For instance, if the two clauses whose lgθ is to be computed were

sort([A,B,C],[B,A,C]) ← C≥A, A≥B (c1)
sort([D,E,F],[E,D,F]) ← F≥D, D≥E (c2)

then the clause lgθ of these two clauses would be:

sort([A,B,C],[B,A,C]) ← C≥A, D≥E, F≥G, A≥B

After reducing (see Definition 2.1) this clause, we would obtain the resulting clause

sort([A,B,C],[B,A,C]) ← C≥A, A≥B

If we write these two clauses in the form of two terms, i.e.

if(sort([A,B,C],[B,A,C]),and(C≥A,A≥B))
if(sort([D,E,F],[E,D,F]),and(F≥D,D≥E)) (t2)

and then take their term lgθ, the resulting term would be

if(sort([A,B,C],[B,A,C]),and(C≥A,A≥B))

where this lgθ corresponds to the clause obtained after taking the (reduced) clause lgθ of the two clauses c1 and c2.

Now, suppose that we change the order of the literals in the body of the clause c2, e.g.

sort([A,B,C],[B,A,C]) ← C≥A, A≥B
sort([D,E,F],[E,D,F]) ← D≥E, F≥D

and compute their clause lgθ, i.e.

sort([A,B,C],[B,A,C]) ← D≥E, C≥A, A≥B, F≥G

After reducing this clause, we obtain the same clause that was computed above when the order of the literals was not changed:

sort([A,B,C],[B,A,C]) ← C≥A, A≥B (c3)

However, if we make this order change for term t2, and then take the term lgθ of the resulting terms, then we obtain the following term

if(sort([A,B,C],[B,A,C]),and(D≥E,F≥G))

where this lgθ does not correspond to the lgθ for clauses, i.e. c3. As we can see, this term is different from the one where the order has not been changed. So, changing the order of the terms matters when term lgθ is used, though it should not matter. Because of that reason, DIALOGS-II uses clause lgθ instead of term lgθ; in that way it also guarantees that there are no second-order lgθs. For instance, suppose that the two clauses, i.e.

delOdds([A,B],[A,B]) ← even(A), even(B)
delOdds([A],[A]) ← even(A)

are given and their clause lgθ is computed as

delOdds([A|B],[A|B]) ← even(A), even(C)

Note that there is no second order variable in the clause lgθ of these two clauses. However, if we write these two clauses in the form of two terms, i.e.

if(delOdds([A,B],[A,B]),and(even(A),even(B)))
if(delOdds([A],[A]),even(A))

and then take their term lgθ, the resulting term would be

if(delOdds([A|B],[A|B]),V)

where the variable V is a second-order variable.

DIALOGS-II Eliminates Redundant Answers. Another new concept related to the queries of DIALOGS-II is “elimination of redundant answers”. Before discussing this concept, we introduce some terminology. We assume that conjunctions of literals can also be viewn as sets of literals.

Definition 3.1: A conjunction of literals C_1 θ-*subsumes* a conjunction of literals C_2 (denoted $C_1 \geq C_2$) iff there exists a substitution σ such that $C_2\sigma \subseteq C_1$.

For instance, let C_1 be B=[C], C=A and C_2 be B=[A]. The conjunction C_1 θ-subsumes C_2 since there exists a substitution σ, which is {A/C}, such that $C_2\{A/C\} \subseteq C_1$.

Theorem 3: $(C_1 \geq C_2) \Rightarrow (C_1 \Rightarrow C_2)$

Proof 3: From C_1, we can build a clause, namely $\neg C_1$. From C_2, we can build a clause, namely $\neg C_2$. Now, note that $C_1 \Rightarrow C_2$ is equivalent to $\neg C_2 \Rightarrow \neg C_1$. So, to check for $C_1 \Rightarrow C_2$, one may approximate this (correctly but incompletely) by checking for $\neg C_2 \geq \neg C_1$ (according to Plotkin's definition, i.e. Definition 2.1, for clauses) (since $\neg C_1$ and $\neg C_2$ are clauses), i.e. by finding a substitution σ such that $\neg C_2\sigma \subseteq \neg C_1$, which is obviously equivalent to $C_2\sigma \subseteq C_1$.

When, to a query (i.e. atom) Q, the specifier gives a DNF answer $C_1 \vee C_2 \vee \ldots \vee C_n$ ($n \geq 0$), then the system must eliminate those C_i for which there exists j such that $C_i \geq C_j$ ($i \neq j$) (i.e. eliminate those that are more general than some other one), and then only build the clauses $Q \leftarrow C_k$, where k is in the set of remaining indices.

What happens when the system does not eliminate redundant answers? We illustrate this point by means of a case that occurs during the synthesis of a program for efface(E,L,R), where efface(E,L,R) holds iff list R is list L without the first (existing) occurrence of term E in L. Let the query and its answer be:

`When does efface(A,[B,A],C) hold?` C=[A], B=A; C=[B].

The system would abduce the following shortcuts and evidence from this answer:

compose_efface(B,[],C,A) ← C=[A], B=A
compose_efface(B,[],C,A) ← C=[B]
solve_efface(A,[B,A],C) ← C=[A], B=A
solve_efface(A,[B,A],C) ← C=[B]
efface(A,[B,A],C) ← C=[A], B=A (s1)
efface(A,[B,A],C) ← C=[B] (s2)

Next, the system generates the query

`When does efface(A,[B,A,A],C) hold?`

where the answer to the query is:

C=[A,A], A=B; C=[B,A], B≠A.

Using the answer, the system would abduce the following shortcuts and evidence:

compose_efface(B,[A],C,A) ← C=[A,A], A=B (c4)
compose_efface(B,[A],C,A) ← C=[B,A], B≠A (c5)
efface(A,[B,A,A],C) ← C=[A,A], A=B
efface(A,[B,A,A],C) ← C=[B,A], A≠B

Upon backtracking to shortcut s2, the system would also abduce the following evidence:

compose_efface(B,[A],C,A) ← C=[A,A], A=B (c6)
compose_efface(B,[A],C,A) ← C=[B,A], B≠A (c7)

Upon backtracking, the following evidence for solve_efface would be abduced using the answer to the query:

solve_efface(A,[B,A,A],C) ← C=[A,A], A=B
solve_efface(A,[B,A,A],C) ← C=[B,A], A≠B

Now, note that c4 and c6 are identical, as well as c5 and c7. This redundancy in the evidence clauses is due to the redundancy in the answer to the query asked for efface(A,[B,A],C). There are now two more compose_efface clauses than solve_efface clauses. This means that in the resulting set of evidence clauses that is passed to the Program Closing Method, there will be more compose_efface clauses than solve_efface, which makes the Program Closing Method fail, because the division algorithm of the Program Closing Method (see Algorithm 8) works under the assumption that there are less compose_efface clauses than solve_efface clauses. This is a correct assumption since there should always be more number of solve_r clauses than the number of compose_r clauses, if the evidence is correctly abduced. This is due to the existence of a decomposition operator in the recursive clause, which does not resolve for some values of the induction parameter, e.g. [ ] for lists, 0 for natural numbers, which in turn causes less evidence to be abduced for the open relation of the recursive clause than the one of the non-recursive clause.

Thus, the system must eliminate the answer C=[A], B=A (which is more general than C=[B]) from C=[A], B=A; C=[B]. So, C=[A], B=A is redundant and is eliminated from the answer, leaving only C=[B] as the answer to the query, where this elimination prevents the redundancy in the evidence clauses, which in turn makes the system to abduce a usable set of evidence clauses.

Chapter 4

Conclusion

The inductive synthesis of recursive (logic) programs is a challenging and important sub-field of ILP. Challenging because recursive programs are particularly delicate mathematical objects that must be designed with utmost care. Important because recursive programs (for certain predicates) are sometimes the only way to complete the induction of a finite hypothesis (involving these predicates).

When it comes to programming applications, we believe that the ideal technique is interactive (in the sense of DIALOGS [13]) and non-incremental, has a clausal evidence language plus type, mode, and multiplicity information (like SYNAPSE [11], DIALOGS), can handle semantic manipulation relations, actually uses (structured) background knowledge and a syntactic bias, which are both problem-independent and intensional (like in SYNAPSE), is guided by (and not just based on) at least the powerful divide-and-conquer schema of SYNAPSE and DIALOGS (using the implementation approach of METAINDUCE [18]), discovers additional base case and recursive case examples (like CILP [19]), can perform both necessary and useful predicate invention (like SYNAPSE, DIALOGS), even from sparse abduced evidence (like CILP), actually discovers the recursive atoms, and makes a constructive usage of the negative evidence (through abduction, like the *Constructive Interpreter* [9] and SYNAPSE).

Thus, we aimed to design and implement a synthesizer that induces recursive logic programs, which is non-incremental, schema-guided, and interactive, and finally developed DIALOGS-II, which is based on the system DIALOGS [13].

DIALOGS-II is a schema-guided, interactive, and non-incremental synthesizer of recursive logic programs that takes the initiative and queries a (possibly naive) specifier for evidence in her/his conceptual language. DIALOGS-II only asks for the minimal knowledge a specifier *must* have in order to want a (logic) program, and it can be used by any learner (including itself) that detects, or merely conjectures, the necessity of invention of a new predicate. Moreover, due to its powerful codification of "recursion-theory" into schemata and schematic constraints, it needs very little evidence and is very fast.

The main difference between DIALOGS-II and its ancestor DIALOGS is as follows: DIALOGS-II enables the specifier to select a certain schema together with a strategy, whereas DIALOGS does not have such a concept of selection of a schema and its strategy, i.e. the concept of schema-guidedness; indeed, DIALOGS is schema-based (has a hard-wired divide-and-conquer schema together with a strategy). To the best of our knowledge, DIALOGS-II is the first schema-guided synthesizer.

Other differences are that DIALOGS-II uses the open program approach (a first-order approach) to representing schemas, whereas DIALOGS uses a second-order approach to representing its divide-and-conquer schema. Using the open program approach simplifies the representation and manipulation of the schemas of the system.

DIALOGS-II handles the sparseness problem, thus enabling the system to induce programs that were not inducable by DIALOGS, e.g. for `factorial`.

DIALOGS uses term lgθs in its MSG Method, whereas DIALOGS-II uses clause lgθs in its Program Closing Method, since clause lgθ is a more powerful way of handling generality among clauses.

DIALOGS-II can induce correct recursive logic programs from less evidence than other ILP systems, e.g. FOIL [24] and TRACY [3]. Moreover, the divide-and-conquer schemata that can be used by the system may be more general than the ones of some other important ILP systems, e.g. CILP [19], CRUSTACEAN [1] [2], and METAINDUCE [18].

DIALOGS-II can be further improved in several ways: a heuristic for the necessary predicate invention would conjecture when to do predicate invention, and finding more powerful admissibility criteria for the evidence of the open relations of the divide-and-conquer schema would increase the probability of synthesizing a correct program.

References

[1] D.W. Aha, S. Lapointe, C.X. Ling, and S. Matwin. Inverting implication with small training sets. In F. Bergadano and L. De Raedt (eds), *Proc. of ECML*'94, pp. 31–48. LNAI 784, Springer-Verlag, 1994.

[2] D.W. Aha, S. Lapointe, C.X. Ling, and S. Matwin. Learning recursive relations with randomly selected small training sets. In W.W. Cohen and H. Hirsh (eds), *Proc. of ICML*'94. Morgan Kaufmann, 1994.

[3] F. Bergadano and D. Gunetti. Learning clauses by tracing derivations. In S. Wrobel (ed), *Proc. of ILP*'94, pp. 11–29. GMD-Studien Nr. 237, Sankt Augustin (Germany), 1994.

[4] A.W. Biermann. Dealing with Search. In W. Biermann, G. Guiho, and Y. Kodratoff (eds), *Automatic Program Construction Techniques*, pp. 375–392. Macmillan, 1984.

[5] H. Büyükyıldız. *Schema-based Logic Program Transformation*. M.Sc. Thesis, Bilkent University, Department of Computer Science, 1997.

[6] H. Büyükyıldız and P. Flener. *Generalized Generalization Generalizers*. In N. E. Fuchs (ed), *Proc. of LOPSTR*'97. LNCS, Springer-Verlag, forthcoming.

[7] W.W. Cohen. Compiling prior knowledge into an explicit bias. In P. Edwards and D. Sleeman (eds), *Proc. of ICML*'92, pp. 102–110. Morgan Kaufmann, 1992.

[8] W.W. Cohen. PAC-learning a restricted class of recursive logic programs. In S. Muggleton (ed), *Proc. of ILP*'93, pp. 73–86. TR IJS-DP-6707, J. Stefan Institute, Ljubljana (Slovenia), 1993.

[9] N. Dershowitz and Y.-J. Lee. Logical debugging. *Journal of Symbolic Computation, Special Issue on Automatic Programming* 15(5–6):745–773, May/June 1993.

[10] E. Erdem and P. Flener. A redefinition of least generalizations and its application to inductive logic program synthesis. In preparation.

[11] P. Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publishers, 1995.

[12] P. Flener. *Predicate Invention in Inductive Program Synthesis*. TR BU-CEIS-9509, Bilkent University, Ankara, Turkey, 1995.

[13] P. Flener. *Inductive logic program synthesis with DIALOGS*. In S. Muggleton (ed), *Proc. of ILP*'96. LNAI, Springer-Verlag, 1997.

[14] P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation, Special Issue on Automatic Programming* 15(5-6):775–805, May/June 1993.

[15] P. Flener, K.-K. Lau, and M. Ornaghi. *On Correct Program Schemas*. In N. E. Fuchs (ed), *Proc. of LOPSTR*'97. LNCS, Springer-Verlag, forthcoming.

[16] P. Flener and L. Popelínský. On the use of inductive reasoning in program synthesis: Prejudice and prospects. In L. Fribourg and F. Turini (eds), *Joint Proc. of META'*94 *and LOPSTR'*94, pp. 69–87. LNCS 883, Springer-Verlag, 1994.

[17] P. Flener and S. Yılmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. Submitted to the *Journal of Logic Programming*.

[18] A. Hamfelt and J. Fischer Nilsson. Inductive metalogic programming. In S. Wrobel (ed), *Proc. of ILP*'94, pp. 85–96. GMD-Studien Nr. 237, Sankt Augustin (Germany), 1994.

[19] S. Lapointe, C. Ling, and S. Matwin. Constructive inductive logic programming. In S. Muggleton (ed), *Proc. of ILP*'93, pp. 255–264. TR IJS-DP-6707, J. Stefan Institute, Ljubljana (Slovenia), 1993.

[20] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proc. of ICML*'88, pp.339–352. Morgan Kaufmann, 1988.

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

[22] S.H. Nienhuys and R. de Wolf. Least generalizations and Greatest Specializations of Sets of Clauses. *Journal of Artificial Intelligence Research* 4: 341–363, 1996.

[23] G. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie (eds), *Machine Intelligence* 5:153–163. Elsevier North Holland, New York, 1970.

[24] J. R. Quinlan and R. M. Cameron-Jones. Induction of logic programs: FOIL and related systems. *New Generation Computing*, pp. 287–312, 1995.

[25] I. Stahl. *Predicate Invention in ILP: An Overview.* TR 1993/06, Fakultät Informatik, Universität Stuttgart (Germany), 1993.

[26] B. Tausend. A unifying representation for language restrictions. In S. Muggleton (ed). *Proc. of ILP*'93, pp. 205–220. TR IJS-DP-6707, J. Stefan Institute, Ljubljana (Slovenia), 1993.

[27] R. Wirth and P. O'Rorke. Constraints for predicate invention. In S. Muggleton (ed), *Inductive Logic Programming*, pp. 299–318. Volume APIC-38, Academic Press, 1992.

Appendix A: README file for DIALOGS-II

After loading the file dialogsII.pl, start a new synthesis by typing “d2.” (without the quotes). The system is composed of the following programs:

- phase0.pl: asks for predicate declaration, schema and strategy, and executes the strategy in order to obtain an open program to be passed to the next phase.
- phase1and2.pl: abduces evidence, induces program clauses by the Program Closing Method and evaluates the result of the Program Closing Method to conjecture necessary predicate invention.
- schemas.pl: contains the currently available schemata and the strategies of the system.
- cliques.pl: finds (admissible) cliques of clauses.
- clausemsg.pl: computes the lgθ of two clauses.
- primitives.pl: contains primitives used by the system.
- grammar.pl: contains Definite Clause Grammar for parsing predicate declarations.
- utilities.pl: contains procedures frequently used by the system
- dedotify.pl: dedotifies initial schemata of the system to convert them to open programs

Variable names start with an uppercase letter; predicate names, functor and constants start with a lowercase letter. Conjunction is expressed by a comma (,), disjunction by a semi-colon (;), negation by wrapping the atom with a prefix neg/1 functor, truth by “true”, and falsity by “false” (without the quotes). The available primitives are: =/2, \==/2, length/2, append/3, member/2, nat/1, list/1, add/3, mult/3, lt/2, gt/2, le/2, ge/2, partition/4, and halves/3 (see file primitives.pl). Natural numbers should be typed in as Peano numbers, using 0 for zero and prefix functor s/1 for successor.

Please note that during the determination of the predicate declaration, parameter roles and decomposition operator, answers should not be terminated by a full-stop (.). The default answer (always between curly braces) can be selected by simply hitting the RETURN/ENTER key. You can force backtracking to a previous question using the answer “back” (without the quotes). Note that parameters that can be any number of (e.g. passive parameters) are indicated as lists, using the Prolog notation; that means the absence of such parameters is indicated using the empty list ([]). For the schema language please refer to [5]. A new schema can be added to the system using that schema language. You also need to make sure that the parameter roles of the parameters of the programs that fit to the schema, modes of the open relations, and the positions of the parameters inside the atoms of the open relations are defined (see file schemas.pl). Available types are atom, term, nat, int, list(atom), list(term), list(nat), and list(int). The type language can be inferred by looking at file grammar.pl (see non-terminal type/1). Similarly for the predicate declaration language. You can express your boredom with the questions (or unwillingness or inability to answer them) by answering “stop_it” (without the quotes). You will find some sample syntheses in the remainder of the thesis.

Appendix B: Sample Syntheses

- len(L,N) iff natural number N is the length of the list L.

```
Predicate declaration? len(L:list(term),N:nat)
Schema? [dc,dg] dc
Strategy? [divide_and_conquer_strategy1]
divide_and_conquer _strategy1
Induction parameter? {L} L
Result parameter? {N} N
Decomposition Operator? {decompose(L,HL,TL)<--L=[HL|TL]}
decompose(L,HL,TL)<--L=[HL|TL]

When does len([],A) hold?
A=0.
When does len([A],B) hold?
B=s(0).
When does len([A,B],C) hold?
C=s(s(0)).
When does len([A,B,C],D) hold?
D=s(s(s(0))).
When does len([A,B,C,D],E) hold?
stop_it.

Result of the Program Closing Method:

Clauses for compose_len:
compose_len(A,B,s(B)) <--

Clauses for solve_len:
solve_len([],0) <--

Please evaluate the Program Closing Method results: need for recursive synthesis? [yes/no] no
A possible program is:
```

```
len(A,B) <-- solve_len(A,B)
len(A,C) <-- decompose_len(A,D,E),len(E,F),compose_len(D,F,C)
decompose_len(G,H,I) <-- G=[H|I]
solve_len([],0) <--
compose_len(J,K,s(K)) <--
Do you want another logic program? {yes} yes

Decomposition Operator? {decompose(L,HL1,HL2,TL)<--L=[HL1,HL2|TL]}
decompose(L,HL1,HL2,TL)<--L=[HL1,HL2|TL]

Using shortcut(s) instead of querying...
When does len([A,B,C,D],E) hold?
E=s(s(s(s(0)))).
When does len([A,B,C,D,E],F) hold?
stop_it.

Result of the Program Closing Method:

Clauses for compose_len:
compose_len(A,B,C,s(s(C))) <--

Clauses for solve_len:
solve_len([A],s(0)) <--
solve_len([],0) <--

Please evaluate the Program Closing Method results: need for recur-
sive synthesis? [yes/no] no
A possible program is:
len(A,B) <-- solve_len(A,B)
len(A,C) <-- decompose_len(A,D,E,F),len(F,G),compose_len(D,E,G,C)
decompose_len(H,I,J,K) <-- H=[I,J|K]
solve_len([],0) <--
solve_len([L],s(0)) <--
compose_len(M,N,P,s(s(P))) <--
Do you want another logic program? {yes} no
No (more) programs.
```

- count(A,B,C) iff natural number C is the number of elements that unify with the term A in list B.

```
Predicate declaration? count(A:term,B:list(term),C:nat)
Schema? [dc,dg] dc
Strategy? [divide_and_conquer_strategy1]
divide_and_conquer _strategy1
Induction parameter? {B} B
Result parameter? {C} C
Passive parameter(s)? {[A]} [A]
Decomposition Operator? {decompose(B,HB,TB)<--B=[HB|TB]}
decompose(B,HB,TB)<--B=[HB|TB]

When does count(A,[],B) hold?
B=0.
When does count(A,[B],C) hold?
C=0,A\==B;C=s(0),A=B.
When does count(A,[B,A],C) hold?
C=s(0),A\==B;C=s(s(0)),A=B.
When does count(A,[B,A,A],C) hold?
C=s(s(0)),A\==B;C=s(s(s(0))),A=B.
When does count(A,[B,A,A,A],C) hold?
stop_it.

Result of the Program Closing Method:

Clauses for compose_count:
compose_count(A,B,s(B),A) <--
compose_count(C,D,D,E) <-- E\==C

Clauses for solve_count:
solve_count(A,[],0) <--

Please evaluate the Program Closing Method results: need for recur-
sive synthesis? [yes/no] no
A possible program is:
count(A,B,C) <-- solve_count(A,B,C)
count(A,D,E) <-- decompose_count(D,F,G),
count(A,G,H),compose_count(F,H,E,A)
decompose_count(I,J,K) <-- I=[J|K]
solve_count(L,[],0) <--
```

```
compose_count(M,N,N,P) <-- P\==M
compose_count(Q,R,s(R),Q) <--
Do you want another logic program? {yes} no.
No (more) programs.
```

- addlast(A,B,C) iff list C is list B with the term A added in the end.

```
Predicate declaration? addlast(A:term,B:list(term),C:list(term))
Schema? [dc,dg] dc
Strategy? [divide_and_conquer_strategy1]
divide_and_conquer _strategy1
Induction parameter? {B} B
Result parameter? {C} C
Passive parameter(s)? {[A]} [A]
Decomposition Operator? {decompose(B,HB,TB)<--B=[HB|TB]}
decompose(B,HB,TB)<--B=[HB|TB]

When does addlast(A,[],B) hold?
B=[A].
When does addlast(A,[B],C) hold?
C=[B,A].
When does addlast(A,[B,C],D) hold?
D=[B,C,A].
When does addlast(A,[B,C,D],E) hold?
E=[B,C,D,A].
When does addlast(A,[B,C,D,E],F) hold?
stop_it.

Result of the Program Closing Method:

Clauses for compose_addlast:
compose_addlast(A,[B|C],[A,B|C],D) <--

Clauses for solve_addlast:
solve_addlast(A,[B],[B|A]) <--
solve_addlast(C,[],[C]) <--

Please evaluate the Program Closing Method results: need for recur-
sive synthesis? [yes/no] no
```

```
A possible program is:
addlast(A,B,C) <-- solve_addlast(A,B,C)
addlast(A,D,E) <-- decompose_addlast(D,F,G),
addlast(A,G,H),compose_addlast(F,H,E,A)
decompose_addlast(I,J,K) <-- I=[J|K]
solve_addlast(L,[],[L]) <--
solve_addlast(M,[N],[N|M]) <--
compose_addlast(P,[Q|R],[P,Q|R],S) <--
Do you want another logic program? {yes} no.
No (more) programs.
Do you want another synthesis with a different strategy? {yes} yes
There is no other strategy for schema dc!
```

- multiply(A,B,C) iff natural number C is the product of natural numbers A and B.

```
Predicate declaration? multiply(A:nat,B:nat,C:nat)
Schema? [dc,dg] dc
Strategy? [divide_and_conquer_strategy1]
divide_and_conquer _strategy1
Induction parameter? {A} A
Result parameter? {B} B
Passive parameter(s)? {[C]} [C]
Decomposition Operator? {decompose(A,HA,TA)<--A=s(TA),HA=A}
decompose(A,HA,TA)<--A=s(TA),HA=A

When does multiply(0,A,B) hold?
A=0.
When does multiply(s(0),A,B) hold?
A=B+0.
When does multiply(s(s(0)),A,B) hold?
A=B+(B+0).
When does multiply(s(s(s(0))),A,B) hold?
A=B+(B+(B+0)).
When does multiply(s(s(s(s(0)))),A,B) hold?
stop_it.

Result of the Program Closing Method:

Clauses for compose_multiply:
```

```
compose_multiply(s(A),B,C+B,C) <--

Clauses for solve_multiply:
solve_multiply(0,0,A) <--

Please evaluate the Program Closing Method results: need for recur-
sive synthesis? [yes/no] no
A possible program is:
multiply(A,B,C) <-- solve_multiply(A,B,C)
multiply(A,D,E) <-- decompose_multiply(A,F,G),
multiply(G,H,E),compose_multiply(F,H,D,E)
decompose_multiply(I,J,K) <-- I=s(K),J=I
solve_multiply(0,0,L) <--
compose_multiply(s(M),N,P+N,P)<--
Do you want another logic program? {yes} no.
No (more) programs.
```

- compress(L,R) iff list R is the compressed form of list L.

  e.g. compress([a,a,b,c,c,c,d],[a,s(s(0)),b,s(0),c,s(s(s(0))),d,s(0)])

```
Predicate declaration? compress(L:list(atom),R:list(atom))
Schema? [dc,dg] dc
Strategy? [divide_and_conquer_strategy1]
divide_and_conquer_strategy1
Induction parameter? {L} L
Result parameter? {R} R
Decomposition Operator? {decompose(L,HL,TL)<--L=[HL|TL]}
decompose(L,HL,TL)<--L=[HL|TL]

When does compress([],A) hold?
A=[].
When does compress([A],B) hold?
B=[A,s(0)].
When does compress([A,B],C) hold?
C=[A,s(s(0))],eq(A,B);C=[A,s(0),B,s(0)],diff(A,B).
When does compress([A,B,C],D) hold, assuming eq(B,C)?
D=[A,s(s(s(0)))],eq(A,B);D=[A,s(0),B,s(s(0))],diff(A,B).
When does compress([A,B,C],D) hold, assuming diff(B,C)?
D=[A,s(s(0)),C,s(0)],eq(A,B);D=[A,s(0),B,s(0),C,s(0)],diff(A,B).
```

```
When does compress([A,B,C,D],E) hold, assuming eq(B,C),eq(C,D)?
stop_it.

Result of the Program Closing Method:

Clauses for compose_compress:
compose_compress(A,[B,s(C)|D],[A,s(0),B,s(C)|D]) <-- diff(A,B)
compose_compress(E,[F,s(G)|H],[E,s(s(G))|H]) <-- eq(E,F)

Clauses for solve_compress:
solve_compress([A],[A,s(0)]) <--
solve_compress([],[]) <--

Please evaluate the Program Closing Method results: need for recur-
sive synthesis? [yes/no] no

A possible program is:
compress(A,B) <-- solve_compress(A,B)
compress(A,C) <-- decompose_compress(A,D,E),
compress(E,F),compose_compress(D,F,C)
decompose_compress(G,H,I) <-- G=[H|I]
solve_compress([],[]) <--
solve_compress([J],[J,s(0)]) <--
compose_compress(K,[L,s(M)|N],[K,s(s(M))|N]) <-- eq(K,L)
compose_compress(P,[R,s(Q)|S],[P,s(0),R,s(Q)|S]) <-- diff(P,R)
```

- s(L,S) iff list S is (ascendingly) sorted version of list L.

```
Predicate declaration? s(L:list(int),S:list(int))
Strategy? [divide_and_conquer_strategy1]
divide_and_conquer _strategy1
Induction parameter? {L} L
Result parameter? {S} S
Decomposition Operator? {decompose(L,HL,TL)<--L=[HL|TL]}
decompose(L,HL,TL)<--L=[HL|TL]

When does s([],A) hold?
A=[].
When does s([A],B) hold?
```

```
B=[A].
When does s([A,B],C) hold?
C=[A,B],le(A,B);C=[B,A],gt(A,B).
When does s([A,B,C],D) hold, assuming le(B,C)?
D=[A,B,C],le(A,B);D=[B,A,C],gt(A,B),le(A,C);D=[B,C,A],gt(A,B),gt(A,C).
When does s([A,B,C,D],E) hold, assuming le(B,C),le(C,D)?
stop_it.

Result of the Program Closing Method:

Clauses for compose_s:
compose_s(A,[B|C],[A,B|C]) <-- le(A,B)

Clauses for solve_s:
solve_s(A,A) <--
solve_s([B,C,D],[C,D,B]) <-- gt(B,C),gt(B,D),le(C,D)
solve_s([E,F],[F,E]) <-- gt(E,F)

Please evaluate the Program Closing Method results: need for recur-
sive synthesis? [yes/no] yes
Need for recursive synthesis detected!
Calling DIALOGS-II with the predicate declaration
   compose_s(HL:int,TS:list(int),S:list(int))
Induction parameter? {[TS]} [TS]
Result parameter? {S} S
Passive parameter(s)? {[HL]} [HL]
Decomposition Operator? {decompose(TS,HTS,TTS)<--TS=[HTS|TTS]}
decompose(TS,HTS,TTS)<--TS=[HTS|TTS]
Current program:
s(A,B) <-- solve_s(A,B)
s(A,C) <-- decompose_s(A,D,E),s(E,F),compose_s(D,F,C)
decompose_s(G,H,I) <-- G=[H|I]
solve_s([],[]) <--
compose_s(J,K,L) <-- solve_compose_s(J,K,L)
compose_s(J,M,N) <-- decompose_compose_s(M,P,Q),
compose_s(J,Q,R),compose_compose_s(P,R,N,J)
decompose_compose_s(S,T,U) <-- S=[T|U]

When does s([A,B,C],D) hold, assuming le(B,C),le(A,C)?
```

```
stop_it.

Result of the Program Closing Method:

compose_compose_s clauses:
(none)

solve_compose_s clauses:
solve_compose_s(A,[B],[B,A]) <-- gt(A,B)
solve_compose_s(C,[D],[C,D]) <-- le(C,D)
solve_compose_s(E,[],[E]) <--

Please evaluate the Program Closing Method results: need for recur-
sive synthesis? [yes/no] no
A possible program is:
s(A,B) <-- solve_s(A,B)
s(A,C) <-- decompose_s(A,D,E),s(E,F),compose_s(D,F,C)
decompose_s(G,H,I) <-- G=[H|I]
solve_s([],[]) <--
compose_s(J,K,L) <-- solve_compose_s(J,K,L)
compose_s(J,M,N) <-- decompose_compose_s(M,P,Q),
compose_s(J,Q,R),compose_compose_s(P,R,N,J)
decompose_compose_s(S,T,U) <-- S=[T|U]
solve_compose_s(V,[],[V]) <--
solve_compose_s(W,[X],[W,X]) <-- le(W,X)
solve_compose_s(Y,[Z],[Z,Y]) <-- gt(Y,Z)
Do you want another logic program? {yes} no
```

- reverse(A,B,C) iff list B is the concatenation of reverse of list A and the list C itself.

```
Predicate declaration? reverse(A:list(term),R:list(term),L:list(term))
Schema? [dc,dg] dg
Strategy? [descend_gen_strategy1]
descend_gen_strategy1
Induction parameter? {A} A
Result parameter? {R} R
Passive parameter(s)? {[L]} []
Accumulation parameter(s)? {[L]} [L]
```

```
Decomposition Operator? {decompose(A,HA,TA)<--A=[HA|TA]}
decompose(A,HA,TA)<--A=[HA|TA]

When does reverse([],A,B) hold?
A=B.
When does reverse([A],B,C) hold?
B=[A|C].
When does reverse([A,B],[B|C],D) hold?
[B|C]=[B,A|D].
When does reverse([A,B,C],[C,B|D],E) hold?
[C,B|D]=[C,B,A|E].
When does reverse([A,B,C,D],[D,C,B|E],F) hold?
[D,C,B|E]=[D,C,B,A|F].
When does reverse([A,B,C,D,E],[E,D,C,B|F],G) hold?
stop_it.

Entering the Program Closing Method with the following evidence
solveAccu_reverse evidence:
solveAccu_reverse([],A,A) <--
solveAccu_reverse([B],[B|C],C) <--
solveAccu_reverse([D,E],[E,D|F],F) <--
solveAccu_reverse([G,H,I],[I,H,G|J],J) <--
solveAccu_reverse([K,L,M,N],[N,M,L,K|P],P) <--

extendAccu_reverse evidence:
extendAccu_reverse(A,B,[A|B]) <--
extendAccu_reverse(C,D,[C|D]) <--
extendAccu_reverse(E,F,[E|F]) <--
extendAccu_reverse(G,H,[G|H]) <--

Result of the Program Closing Method:

Clauses for extendAccu_reverse:
extendAccu_reverse(A,B,[A|B]) <--

Clauses for solveAccu_reverse:
solveAccu_reverse([],A,A) <--
```

```
Please evaluate the Program Closing Method results: need for recur-
sive synthesis? [yes/no] no

A possible program is:
reverse(A,B,C) <-- solveAccu_reverse(A,B,C)
reverse(A,D,E) <-- decompose_reverse(A,F,G),
                  reverse(G,D,H),extendAccu_reverse(F,E,H)
decompose_reverse(I,J,K) <-- I=[J|K]
solveAccu_reverse([],L,L)<--
extendAccu_reverse(M,N,[M|N])<--
Do you want another logic program? {yes}
no
```