Inductive Synthesis of Recursive Logic Programs: Achievements and Prospects

Pierre Flener
Department of Information Science
Uppsala University, Box 311
S–751 05 Uppsala, Sweden
Email: pierref@csd.uu.se
http://www.csd.uu.se/~pierref/

Serap Yılmaz
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, USA
Email: syilmaz@microsoft.com

Abstract

The inductive synthesis of recursive logic programs 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. We first overview past and present achievements, focusing on the techniques that were designed specifically for the inductive synthesis of recursive logic programs, but also discussing a few general ILP techniques that can also induce non-recursive hypotheses. Then we analyse the prospects of these techniques in this task, investigating their applicability to software engineering as well as to knowledge acquisition and discovery.

Keywords: inductive inference, inductive logic programming (ILP), recursion, schemas.

1 Introduction

Examples are better than precepts; let me get down to examples—
I much prefer examples to general talk.
— George Polya

In a quite general first formulation, 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 logic programs. We then say that H covers E (in B). In practice, B and H are often restricted to definite logic programs. Evidence E is often further refined into positive evidence E^+ (which is to be covered by the hypothesis) and negative evidence E^- (which is not to be covered by the hypothesis) (sometimes, it is labelled as negative, rather than explicitly negated). Often, the elements 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)
```

then *a* possible hypothesis[1] 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 also do it. 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 (although we will not address this latter issue here).

General surveys of the achievements of ILP exist [7] [49] [59] [78], as well as proceedings of ILP workshops and edited collections of reports on landmark ILP research. In this paper, we more closely and almost exclusively overview the achievements of ILP techniques that have been specifically designed for the induction of *recursively* expressed hypotheses (or simply: recursive hypotheses), such as the subset program above. To be precise, we mean the class of logic programs where at least one clause is recursive (i.e., it has at least one body atom with the same predicate symbol as its head atom). The induction of this class of hypotheses is much harder than the one of non-recursive hypotheses. The fact that one does not in general know in advance whether a recursive hypothesis exists or not seems to speak in favour of only using more general-purpose ILP techniques, i.e., techniques that can induce both recursive and non-recursive hypotheses. However, the study of recursion induction is worthwhile in its own right, and it gives rise to important applications.

Recursive programs actually *compute* something, in the traditional understanding of what a program is and does, but such is not the case with all non-recursive programs, which might for instance *classify* data as belonging to one concept or another [32]. Inferring recursive programs from assumed-to-be-complete information such as the axiomatisation

$$\text{subset(S,L)} \Leftrightarrow \forall \text{X (member(X,S)} \Rightarrow \text{member(X,L))}$$

where member is a known predicate (with the usual meaning), is called *program synthesis*, and features two main approaches, namely deductive synthesis and constructive synthesis.[2] We adopt the synthesis terminology here, and talk of *inductive synthesis* (of recursive programs) from incomplete specifications whenever we want to focus on this sub-field, and of *ILP* when we mean the whole field.

The achievements in the synthesis of (recursive) logic programs, whether by deductive, constructive, inductive, mixed, or even manual techniques, have been surveyed [25], but with only marginal detail on inductive techniques. One purpose of our paper is thus to complement that survey and to specialise the already mentioned general surveys of ILP. Our other purpose is to discuss the prospects of this important sub-field. Although nobody denies its intrinsic interest, there has been considerable debate on its industrial applications. We summarise the existing opinions, debunk or support them when necessary, and bring in a few possibly new considerations.

The rest of this paper is thus organised as follows. First, in Section 2, we introduce some additional terminology and some theoretical results regarding the inductive synthesis of recursive programs, laying the groundwork for a classification of such techniques. Next, in Section 3, we overview the achievements of inductive synthesis, and in Section 4, we discuss its application prospects. Finally, in Section 5, we conclude.

2 Terminology and Theoretical Results

We now introduce some additional terminology (in Section 2.1 and Section 2.2) and mention some theoretical results (in Section 2.3 and Section 2.4) concerning the induction of recursive clauses. This allows us to have classification features for the techniques overviewed in the third section.

1. Note that, contrary to common practice, we do *not* talk about "*the* target hypothesis," as there may be *many* possible hypotheses for a given predicate, especially when, as advocated later, background knowledge, bias, and evidence do not encode (part of) a possible hypothesis.
2. It should be noted that non-recursive (or non-looping) procedures constitute the vast majority of the code of a software application. However, not much research is needed to (semi-)automatically infer non-recursive programs from assumed-to-be-complete formal specifications, as the latter usually already come in non-recursive form. The situation is not quite the same for known-to-be-incomplete formal specifications, and we discuss this issue at the beginning of Section 4.2.1.

2.1 Approaches to ILP (and Inductive Synthesis)

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

2.1.1 Agents

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. For reasons to be discussed in Section 4.2.1, such a machine learning terminology is sometimes misleading, and we shall use the more general terminology of *source*, *induction technique*, and *induction* instead.

2.1.2 Evidence

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 supersets of the intended relations. We here assume that the evidence has *correct* information, i.e., that it describes subsets of the intended relations. In this case, one also says that there is no *noise*. Often, the actually described subsets are finite. An extreme case of incomplete but correct information is complete (i.e., not incomplete) and correct information, though this can often only be achieved through some finite axiomatisation in the hypothesis language, but not in the evidence language (e.g., because the latter does not feature recursion).

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 and select above are syntactic manipulation relations, because they treat different constants like different variables: the atom subset([d,c],[c,e,d]) basically represents the atom subset([D,C],[C,E,D]), where C, D, E are different variables. So the (predefined) equality and inequality predicates suffice to express hypotheses for such relations. However, relation sort embodies a semantic manipulation: the atom sort([2,1],[1,2]) does not represent the atom sort([X,Y],[Y,X]), because otherwise the atom sort([3,4],[4,3]) would be erroneously covered as well. So comparison predicates are needed to express hypotheses for such relations. This raises the question of the discovery of these comparison predicates, and indicates why the induction of hypotheses for semantic manipulation relations is much harder than for syntactic manipulation relations.

We distinguish evidence that can be truly *arbitrary* from evidence that has to be carefully *crafted*. In the latter case, there is a further distinction according to whether the evidence is crafted independently of all possible hypotheses or is made with a possible hypothesis in mind. The same distinction holds for additional input information (see Section 2.2 below) and background knowledge. Obviously, this last category is impractical in most application settings, because then no new knowledge is actually discovered.

Clausal evidence, if not restricted to (positive and negative) examples, is sometimes called *integrity constraints* [22] [43] or *properties* [28] [31], as it constrains hypotheses to satisfy them. This does not affect the statement of the ILP task above, as the most common setting with examples is just a particular case thereof.

2.1.3 Background Knowledge

The background knowledge, although clausal in general, is sometimes restricted to a finite set of ground literals. Such an extensional representation seriously affects practicality in some application settings, such as when the background knowledge must be provided manually for each session. Sometimes, such literals are generated from an intensional clausal representation of the background knowledge before the induction starts. In any case, this extensional representation is used when the verification of the coverage of the evidence by a hypothesis is not based on some form of execution of the hypothesis, or when there are some theoretical limitations.

2.1.4 Induction

Induction (in the sense of ILP, see the introduction) can be viewed as *search* through a graph (or: search space) where the nodes correspond to hypotheses and the arcs correspond to hypothesis-transforming induction operators (or: inductive inference rules). The challenge is to efficiently navigate through such a search space, via intelligent control (e.g., by organising 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 requests for classification of an invented example as a positive or negative one (*classification queries*), requests for instantiation of a variable in an atom so that the atom is a positive example (*instantiation queries*), etc.

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).

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

An *identification criterion* defines the moment where an incremental induction technique has been successful in correctly identifying the intended relations, whether it "knows" this or not. Sample criteria are finite identification, identification-in-the-limit, probably-approximately-correct (PAC) identification (see [19] [20] for algorithms and negative results on PAC-inducing recursive logic programs), and so on (see [59]). There are limiting theorems stating what hypothesis languages are inducable from what evidence languages under what identification criteria [3].

2.1.5 Hypotheses

In the hypothesis, some predicate symbols may be recursively defined: the corresponding clauses are then 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 relations, 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 covering other evidence. The given evidence is thus also called the *training set*, whereas the additional evidence is called the *test set*. We here assume that the test set also has no noise. If evidence is divided into positive and negative evidence, then a hypothesis is *complete* w.r.t. the evidence if it covers all the positive evidence; it is *consistent* w.r.t. the evidence if it does not cover any of the negative evidence; it is *correct* w.r.t. the evidence if it is both complete and consistent. We also use this terminology when comparing to a single piece of evidence. Under an appropriate coverage notion, the same terminology also applies to single clauses rather than hypotheses (which are clause sets). Finally, the same terminology also applies when considering the intended relations instead of the positive evidence, and their complements instead of the negative evidence.

It seems desirable to achieve some separation of concerns regarding the logic and control components of hypotheses when they are 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, to ensure 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 then mentioned in additional inputs, see below). If an interpreter of the target language is actually used during the induction (say, to verify the coverage of the evidence), then such control aspects cannot be entirely ignored while constructing the logic component.

2.2 Extended ILP Settings

It is possible to augment the statement of the ILP task by adding parameters other than the evidence E, the background knowledge B, and the hypothesis H.

One generalisation of the ILP task is known as *theory-guided induction*, or *inductive theory revision*, or *declarative debugging*: 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 variation thereof as possible, in the sense that only the "bugs" of H_i w.r.t. E should be (incrementally) detected, located, and corrected (or: "debugged") in order to produce H. This generalised 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. This is also known as *model-driven* or *approximation-driven* induction, as opposed to *data-driven* induction, where there is no initial theory.

Another generalisation of the ILP task involves augmenting the inputs with *declarative bias*, which is any form of additional input information that restricts the search space [60]. There are three complementary approaches to this, namely *search bias*, *language bias*, and *validation bias* (which gives an acceptance criterion for an incremental induction process, telling when a hypothesis is acceptable; this is related to identification criteria, and may for instance be an accuracy threshold for the test set). We further discuss only the former two here, in the next sections.

2.2.1 Search Bias

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

- Description (*i*) should define the intended relation as declaratively as possible, i.e., without saying how it could possibly be identified. Whether it should be informal or formal is an on-going debate [51], but we do not 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 (namely the evidence) are actually even *assumed* to be incomplete. If restricted to examples, evidence is furthermore a very declarative (formal) description, because it is then impossible to bias towards a possible program.
- Description (*ii*) 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 *determinism* (or: *multiplicity*) 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 [24] [71], it is only natural to provide (some of) it as an additional input to an ILP task, especially for a synthesis task. Such information is thus part of what is called *search bias* (a kind of bias that determines which part of the hypothesis space is searched, and how it is searched). Of course, such information should ideally also be known for all the predicates defined in the background knowledge. We do not discuss other forms of search bias here, and refer the reader to a survey [60].

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 determinism information, namely that the intended relation is a total function in a given mode (i.e., its multiplicity is 1–1). This has a good influence on the amount of negative evidence that has to be explicitly given: if a relation is known to be functional from some parameters to the other parameters, then every atom obtained from a positive example by changing the values of those other parameters is an implicitly given negative example.

2.2.2 Language Bias

Language bias determines the language of hypotheses. One particularly useful and common approach is to bias induction by a schema. Informally, a *program schema* [33] contains a template program and a set of axioms. The *template* abstracts a class of actual programs (called *instances*), in the sense that it represents their dataflow and control-flow by means of *place-holders*, but does not contain (all) their actual computations nor (all) their actual data structures. The *axioms* restrict the possible instances of the place-holders and define their interrelationships. Note that a schema is thus problem-independent. A formal definition of program schemas, and the corresponding representation issues, are beyond the scope of this paper, but they are fully discussed elsewhere [33]. An overview of approaches to program schemas would also take too much space here, so we refer to existing overviews, namely [28] in general, and [60] for ILP approaches only. Let us here take a first-order approach, and consider templates as *open programs* (programs where some predicates — the place-holders — are left undefined, or *open*; a program with no open predicates is said to be *closed*), and axioms as first-order specifications of these open predicates.

Example 1: Let us design a template 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:

r(X,Y) $\leftarrow$ primitive(X), solve(X,Y)
r(X,Y) $\leftarrow$ nonPrimitive(X), decompose(X,H,X_1,X_2), r(X_1,Y_1), r(X_2,Y_2), compose(H,Y_1,Y_2,Y)

The intended semantics 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 values 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. In the latter case, the divide-and-conquer principle is applied by:

(1) division (through decompose) of X into a term H and two terms X_1 and X_2 that are both of the same type as X but smaller than X according to some well-founded relation;

(2) conquering (through r) to determine the value(s) of Y_1 and Y_2 corresponding to X_1 and X_2, respectively;

(3) combining (through compose) terms H, Y_1, Y_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 *any* program is an instance of it (it suffices to instantiate primitive by a program that always succeeds, and solve by the given program). One way to do this is to attach to the template some axioms (omitted here, see [33]), namely the set of specifications of its open predicates: these specifications refer to 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 [33]. Such a schema (i.e., template plus axiom set) constitutes an extremely powerful language bias, because it encodes algorithm design knowledge that would otherwise have to be hardwired or rediscovered the hard way during each synthesis. ♦

The issues in the design and expression of divide-and-conquer logic program schemas are discussed elsewhere in great detail by the first author [28]. Let us here just point out the sub-class of *incomplete traversal programs*, where the induction parameter X need not be entirely traversed before being able to build Y. Programs of this class include the ones for select (as in Section 1) and member. This sub-class seems particularly hard to synthesise: when researchers report pathological relations that elude their synthesisers or require synthesis times disproportionately larger than for other relations that are seemingly of the same level of difficulty, then they are quite often of this sub-class. The reason therefore is the complex *semantic* interplay between primitive and nonPrimitive (note that it is not $\neg$primitive), as it is then not just a *syntactic* distinction of whether the induction parameter is, say, the empty list or a non-empty list, but a *semantic* distinction based on the *values* in the list.

Other approaches to language bias are the clause description language of [5], antecedent description grammars [17], argument dependency graphs [79], etc., and they are surveyed in [7] [60] [76].

2.3 Generality

Given the formula $G \models S$, we say that G is *more general* than S, and that S is *more specific* than G. In our initial formulation of ILP, the objective is to compute a hypothesis H given background knowledge B and (positive) evidence E, such that $B \land H \models E$. The generality relation $\models$ is a partial order, but does not induce a lattice on the set of formulas. Indeed, there is not always a unique least generalisation under implication of an arbitrary pair of clauses. For instance, the clauses p(f(X)) ← p(X) and p(f(f(X))) ← p(X) have both p(f(f(X))) ← p(X) and p(f(X)) ← p(Y) as least generalisations. In [61], the existence and computability of a least generalisation 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 [53], different models of generality have been proposed. We here mainly discuss the generality models that are actually used in the overviewed special-purpose techniques that are dedicated to the inductive synthesis of recursive logic programs, even though they are the weaker models.

2.3.1 θ-subsumption

In the model called θ-subsumption [62] [63], the background knowledge B is empty. The model is defined for clauses, which are here viewed as sets of literals.

Definition 1 (θ-subsumption, θ-subsumption-equivalence, reduced clause)
A clause g θ-*subsumes* a clause s if there exists a substitution σ such that $g\sigma \subseteq s$.
Two clauses are θ-*subsumption-equivalent* if they θ-subsume each other.
A clause is said to be *reduced* if it is not θ-subsumption-equivalent to any proper subset of itself.

Example 2: The clause p(X,Y) ← q(X,Y), r(X) θ-subsumes p(V,Z) ← q(V,Z), q(V,T), r(V), s(Z) with substitution {X/V, Y/Z}. The clause p(V,Z) ← q(V,Z), r(V) is a reduced version of p(V,Z) ← q(V,Z), q(V,T), r(V). ♦

If a clause g θ-subsumes a clause s, then $g \models s$, but the converse is not true for recursive clauses and tautological clauses [63]. For instance, for the recursive clauses p(f(X)) ← p(X) and p(f(f(X))) ← p(X) (called g and s respectively), although $g \models s$ (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 generalisation under* θ*-subsumption* (abbreviated by lgθ) of two clauses c and d, denoted by $lg\theta(c,d)$, is the lub of c and d in the θ-subsumption lattice. A more constructive definition of this operator emerges as a property:

Definition 2 (Least generalisation under θ-subsumption)
The lgθ of two terms $f(s_1,\ldots,s_n)$ and $f(t_1,\ldots,t_n)$, denoted by $lg\theta(f(s_1,\ldots,s_n),f(t_1,\ldots,t_n))$, 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 new variable V, where V represents this pair of terms throughout.
The lgθ of two positive literals $p(s_1,\ldots,s_n)$ and $p(t_1,\ldots,t_n)$, denoted by $lg\theta(p(s_1,\ldots,s_n),p(t_1,\ldots,t_n))$, is $p(lg\theta(s_1,t_1),\ldots,lg\theta(s_n,t_n))$, the lgθ being undefined when the predicate symbols or the arities are different. (Similarly for two negative literals.)
The lgθ of two clauses c and d, denoted by $lg\theta(c,d)$, is $\{lg\theta(l_1,l_2) \mid l_1 \in c \text{ and } l_2 \in d\}$.

Example 3: The lgθ of the clauses p(V,W) ← q(V,W), r(V), s(W) and p(T,N) ← q(T,N), r(T), r(N) is the clause p(X,Y) ← q(X,Y), r(X), r(Z). This clause is reduced. In general, the lgθ of two clauses is not reduced. ♦

2.3.2 Relative θ-subsumption

A first extension of θ-subsumption that uses background knowledge B is called relative subsumption [62].

Definition 3 (Relative θ-subsumption)
If the background knowledge B consists of a finite conjunction (or set) of ground facts, then the *relative least generalisation under* θ*-subsumption* (abbreviated by rlgθ) of two ground atoms E_1 and E_2 relative to background knowledge B is $lg\theta((E_1 \leftarrow B),(E_2 \leftarrow B))$.

Example 4: Given the positive examples e_1 = son(o,a) and e_2 = son(j,t) and the background knowledge B = {parent(a,o), parent(a,t), parent(t,j), parent(t,k), female(a), male(j), male(o)}, the rlgθ of e_1 and e_2 relative to B is:

son(X,Y) ← parent(a,o), parent(a,t), parent(t,j), parent(t,k), female(a), male(j), male(o),
parent(a,Z), parent(Y,V), parent(Y,W), parent(Y,U), parent(t,T), male(X). ♦

The rlgθ of two clauses is not necessarily finite. However, it is possible [59] to construct finite rlgθs under the language bias of *ij*-determinacy [58]:

Definition 4 (Determinacy)
If L_i is a literal in the ordered Horn clause $A \leftarrow L_1,\ldots,L_n$, then the *input variables* of L_i are those appearing in L_i that also appear in the clause $A \leftarrow L_1,\ldots,L_{i-1}$; all other variables in L_i are called *output variables*.
A literal is *determinate* if its output variables may have at most one binding, given a binding of its input variables.
If a variable V appears in the head of a clause, then the *depth* of V is zero; otherwise, if F is the first literal containing the variable V and d is the maximal depth of the input variables of F, then the depth of V is d+1.
A clause is *determinate* if all literals in its body are determinate.
A determinate clause is *ij-determinate* if all literals in its body contain only variables of depth at most i as well as predicate symbols that have arity at most j.

Example 5: The clause p(X,W) ← q(X,W), r(W,Z), p(W,Z) is 32-determinate, provided all literals in its body are determinate. ♦

This model of relative subsumption is restricted to ground background knowledge, but was generalised later to any kind of Horn clausal knowledge [15]. Such generalised subsumption is however not used by any of the techniques overviewed here.

2.3.3 Inverse Resolution

Another model of generality is inverse resolution, based on inverting one or two resolution steps so as to induce some of its/their antecedent(s) from the other antecedent(s) and the consequent(s). There are four inductive inference rules of inverse resolution, namely *absorption*, *identification*, *intra-construction*, and *inter-construction*, given here for propositional logic, but also available for first-order logic [57]:

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

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

Lower-case letters represent atoms, upper-case letters represent conjunctions of atoms. The absorption and identification rules (also known as the *V* rules) invert only one resolution step. The intra-construction and inter-construction rules (also known as the *W* rules) invent new predicate symbols (predicate invention, see the next subsection). Absorption being incomplete, most-specific *V* rules have been introduced [55], as well as a *saturation* rule [67].

2.3.4 Inverse Implication (⇒) and Inverse Entailment (|=)

Recently, a lot of research was undertaken to explore even more powerful models of generality, based on inverting implication [40] or inverting entailment. Since only two of the techniques described here employ this, we do not go into more details here and refer the interested reader to overviews [59] [56].

2.4 Predicate Invention

Nothing is more important than to see the sources of invention,
which are, in my opinion, more interesting than the inventions themselves.
— Gottfried Wilhelm Leibnitz

Predicate invention is the process of introducing into the hypothesis some predicates that are not in the evidence, nor in the background knowledge (this is called shifting the bias by extending the hypothesis language [72]), and then inducing hypotheses for these new predicates. This requires the usage of *constructive* rules of inductive inference (where the inductive consequent may involve predicate symbols that are not in the antecedent), as opposed to *selective* ones (where the inductive consequent can only involve predicate symbols that are in the antecedent). Such constructive induction does not assume that the preliminary tasks of representation choice 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, as discussed next.

2.4.1 Necessary Predicate Invention

We first define necessary predicate invention, and then give an example for it.

Definition 5 (Necessary predicate invention)
Predicate invention is *necessary* if there is no finite hypothesis (satisfying the current bias) for the predicates in the evidence that uses only the predicate symbols in the evidence and in the background knowledge.

Example 6: Assume we want to induce a logic program for the sort predicate (where sort(L,S) holds iff S is a non-decreasingly ordered permutation of integer-list L) from some positive and negative examples, and this in the absence of background knowledge. If, at some moment during induction, the current hypothesis is the following over-general program:

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

then the insert predicate (where insert(E,L,R) holds iff R is non-decreasingly ordered integer-list L with integer E inserted at the "right" place) must necessarily be invented, because there is no other way to complete that hypothesis into a program that covers all the positive examples but none of the negative ones. The resulting overall program could then be the following

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

```
    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 insert necessitated in turn the invention of the ≤ predicate (whose obvious specification and program are omitted here). Also note that the program for insert is recursive: it can thus not be eliminated by unfolding inside the recursive clause for sort. If another recursive clause had been in the over-general hypothesis, then another predicate would have been necessarily invented. Otherwise, the background knowledge being empty, sort would have had 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 (that extensionally encodes the model of sort). Now, even if the background knowledge contained the classical member, length, and append predicates, the invention of insert would still be necessary, because insert cannot be implemented in terms of these background predicates either. ♦

2.4.2 Non-necessary Predicate Invention

We distinguish two types of non-necessary predicate invention, namely *useful* predicate invention and *pragmatic* predicate invention [29].

Definition 6 (Useful and pragmatic predicate invention)
Non-necessary predicate invention is *useful* if the hypothesis for the invented predicate is recursive. Otherwise, it is *pragmatic*.

Example 7: If there were permutation and ordered predicates in the background knowledge of Example 6, then, at the considered moment during induction, the invention of insert such that it is recursively defined (e.g., as above) is useful. But it is not necessary, because insert could then be defined (non-recursively) as follows:

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

This hypothesis for insert could however be eliminated by unfolding inside the recursive clause for sort, which is why the invention was non-necessary. Moreover, this hypothesis for insert would have a complexity of O(n!), where n is the length of the list L, and would thus be very inefficient compared to the recursive insert hypothesis in Example 6, which is O(n). Hence, the induction of that recursive insert hypothesis decreases the complexity of the overall induced sort program, which is why the invention is considered useful, although non-necessary. ♦

Example 8: Given evidence of the grandDaughter relation (where grandDaughter(G,P) holds iff person G is a grand-daughter of person P), and given as background knowledge the parent, female, and male predicates (where parent(P,Q) holds iff person P is a parent of person Q), the induction of the following hypothesis:

```
    grandDaughter(G,P) ← parent(P,Q), daughter(G,Q)
    daughter(D,P) ← parent(P,D), female(D)
```

involved the invention of the daughter predicate (where daughter(D,P) holds iff person D is a daughter of person P). This invention is non-necessary, since the daughter hypothesis can be eliminated by unfolding into the grandDaughter hypothesis, but it also is pragmatic, since it causes the grandDaughter hypothesis to become more compact, and since the daughter concept has now been defined and can be reused in the future. ♦

The reader may wonder whether our non-standard distinction of useful and pragmatic predicate invention is itself a useful invention! At this point, we can already argue that useful predicate invention potentially makes hypotheses much more efficient (as shown in Example 7). Later, in Section 4.2.1, we will argue that the ability of performing useful predicate invention is a major step towards avoiding the background knowledge usage bottleneck (which is basically a state where an induction technique is getting confused by too much background knowledge), and is thus a very desirable feature of induction techniques.

2.4.3 Theoretical Results about Predicate Invention

The task of inductive inference amounts in the limit to finding a finite axiomatisation for a given model. If the intended model cannot be finitely axiomatised within a language $\mathcal{L}$, then inductive inference will never succeed. However, detecting this is undecidable. This follows from Rice's theorem (as initially proven in [72]):

Theorem 1: Given a recursively enumerable, deductively closed set $\mathcal{C}$ of formulas, and the first-order language $\mathcal{L}$ defined from the symbols occurring in $\mathcal{C}$, it is undecidable whether $\mathcal{C}$ is finitely axiomatisable in $\mathcal{L}$ or not.

Therefore, either a heuristic has to be used to conjecture the necessity of predicate invention, or the hypothesis language has to be reduced so that the detection of necessary predicate invention is decidable [73]. Fortunately, introducing new predicate symbols always allows finding a finite axiomatisation, as proved by Kleene (see [72]):

Theorem 2: Any recursively enumerable, deductively closed set $\mathcal{C}$ of formulas in a first-order language $\mathcal{L}$ is finitely axiomatisable using additional predicate symbols not in $\mathcal{L}$.

In other words, Kleene's theorem states that inductive inference will always succeed provided the technique invents the appropriate new predicates. Since only necessary predicates need to be invented, it turns out that necessary predicate invention is crucial in inductive inference. Depending on the hypothesis language, predicate invention is however not always appropriate, because it may be unable to help make the induction succeed [73].

The difficulties of predicate invention are as follows. First, adequate formal *parameters* for the new predicate have to be identified among all the variables in the clause calling that new predicate. This can be done by lengthy computations based on notions such as active (or: discriminating) variables [54], or it can be done instantaneously by using pre-computations done once and for all at the template level [28]. Second, *evidence* of the new predicate has to be abduced from the current hypothesis using the evidence of the old predicate (note that a recursion synthesiser may invoke itself from such abduced evidence). This usually requires an oracle for the old predicate, whose hypothesis is still unfinished at that moment and can thus not be used. Third, the abduced evidence usually is less numerous than for the old predicate (if the new predicate is in the recursive clause, then no new evidence is abduced from the old evidence that is covered by the base clause) and can be quite *sparse*, so that the new synthesis is more difficult. The *sparseness problem* can be illustrated by an example. Given the positive examples factorial(0,1), factorial(1,1), factorial(2,2), factorial(3,6), and factorial(4,24), and given the hypothesis:

```
factorial(0,1) ←
factorial(N,F) ← N=s(M), factorial(M,G), product(N,G,F)
```

where product was just invented (and named so only for the reader's convenience), then the abduced examples are product(1,1,1), product(2,1,2), product(3,2,6), and product(4,6,24), which is hardly enough (note that there is one less example than for factorial) for inducing a recursive hypothesis for product. Indeed, examples such as product(3,6,18), product(2,6,12), product(1,6,6), etc., are missing, which puts the given examples more than one resolution step apart, if not on different resolution paths. This is aggravated by the absence of an oracle for product, because product is not necessarily a concept known to the source of evidence of factorial (remember that it is only called product for convenience, but that, in practice, it has an non-suggestive name). For those techniques that can perform necessary/useful predicate invention, we will discuss how they tackle these difficulties.

3 Achievements of Inductive Synthesis

This section first overviews the achievements of special-purpose techniques that were designed specifically for the inductive synthesis of recursive logic programs. We then overview some representative general-purpose techniques that can induce both recursive and non-recursive logic programs, and explain how they can introduce recursion into a hypothesis. The first *overview* might be incomplete, but it discusses (most of) the landmark techniques in this field. Furthermore, we here only present the techniques (but not their implementations as systems, as the latter may be incomplete) as well as their inputs and outputs, but refrain from judging them in terms of, say, the realism of providing these inputs, as it all depends on the application setting. Any criticism is thus delayed to Section 4.

Our primary classification criterion for the special-purpose synthesis techniques is whether the technique is biased by a program schema or not, which gives rise to Sections 3.1 and 3.2, respectively. In each of these sections, the order of presentation of the techniques is not necessarily chronological, because, other than between the techniques developed at the same institution, there is unfortunately very little influence of techniques on each other. Although the discipline of inductive synthesis of (any kind of) recursive programs is quite old (see [8]), this may be seen as a symptom that a very difficult topic is being tackled here and that very few standard concepts and solutions have appeared yet. It is thus very difficult — if not impossible — to identify a useful generic algorithm covering most of the special-purpose synthesis techniques, and thus to lift this overview to an actual *survey.* Indeed, other than saying that such a generic algorithm would consist of two steps, namely somehow inducing base clauses and somehow inducing recursive clauses, and this in any sequence or even in parallel, there is very little in common to all techniques. This paper is an attempt to bring some order into this state of affairs. Some general-purpose techniques are overviewed in Section 3.3. Finally, in Section 3.4, we point out cross-fertilisation opportunities and identify directions for future work. The comparison chart (Table 1) at the end of this section will be helpful towards this aim, and it may be a good idea for the reader to briefly study it right now.

Techniques that are somehow related to some others, or representative thereof, and techniques that are somehow more sophisticated and powerful (in an absolute, application-independent sense) than others will obviously get more coverage here than those that are completely different from all others, or that feature highly specialised (sub-)machinery that is impossible to explain in the allotted space, or whose power is quite limited. So, and in any case, we refer the reader to the original papers for more details, but we try to keep our assessments independent of unmentioned details.

3.1 Schema-biased Synthesis

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 axiom set, but the schema itself is to a certain degree hardwired into the technique. A useful generalisation is *schema-guided* synthesis, where the schema is either chosen (among already available ones) by the schema-independent technique or provided to it by the source; the schema thus actively guides the synthesis.

A Generic Schema-biased Inductive Synthesis Algorithm. Fortunately, most techniques of schema-biased inductive synthesis *are* amenable to a generic algorithm, which we present next. The templates of the considered schemas are all of the following generic template, which is thus a template template (*sic*):

$r(X,Y,Z) \leftarrow \underline{c}(X,Y,Z),\ p(X,Y,Z)$
$r(X,Y,Z) \leftarrow \underline{d}(X,H,X_1,\ldots,X_t,Y_1,\ldots,Y_t,Z),\ r(X_1,Y_1,Z),\ \ldots,\ r(X_t,Y_t,Z),\ q(X,H,X_1,\ldots,X_t,Y_1,\ldots,Y_t,Z)$

where p and q are two open predicates, whereas $\underline{c}$ and $\underline{d}$ summarise the effect of possibly empty sequences of open predicates (whose joint dataflow here is only indicative). Also, X is a non-empty sequence of terms, whereas Y and Z are possibly empty sequences of terms. Note that the actual sequence of body atoms in the clauses is irrelevant, and that there are thus possibly several ways of obtaining the template of Example 1, say (for instance, q could be either nonPrimitive, or decompose, or compose). Finally, this covers more than just divide-and-conquer programs, so that other design methodologies can be captured in other templates fitting this generic template; for instance, the classical non-naive program for reverse (with an accumulator, thus) is covered, although it is not a divide-and-conquer program.

Now we can introduce our generic algorithm. It is restricted to single-predicate evidence, but it can handle clausal evidence (rather than just examples), and has necessary/useful predicate invention built into it:

Generic Algorithm *SchemaBiasedInductiveSynthesis* (abbreviated sbis)
Inputs:
- a clausal evidence set E_r for some predicate r
- an oracle O_r for r (this may or may not be E_r)
- a context program C in which only r is undefined (this may be the empty program)

Outputs:
- a closed program P_r (containing C) for r that covers E_r in the available background knowledge

Algorithm:

```
sbis($E_r$,$O_r$,C,$P_r$) ←
    selectSchema(S),           % S is (a suitable renaming for r of) some schema
    close-$\underline{cd}$(S,C,V),           % V = C ∪ T ∪ $\underline{CD}$, where T is the template of S, and $\underline{CD}$ defines $\underline{c}$ and $\underline{d}$
    abduce(V,$E_r$,$O_r$,$E_p$,$E_q$),    % $E_p$ (resp. $E_q$) is a clausal evidence set of p (resp. q) abduced by
                               % attempting to prove, using $O_r$, that V covers $E_r$
    induce($E_p$,$E_q$,$P_p$,$P_q$),       % $P_p$ (resp. $P_q$) is a non-recursive program for p (resp. q)
    acceptable($P_q$) →            % (heuristically) decide whether predicate invention is necessary/useful
        $P_r$ = V ∪ $P_p$ ∪ $P_q$        % no need for predicate invention: assemble and return $P_r$
      ; recurse($E_r$,$E_p$,$E_q$,$O_r$,V,$P_p$,$P_q$,$P_r$)   % perform necessary/useful predicate invention by using sbis
```

This generic algorithm sbis is meant to be called whenever there are good reasons to believe that a new predicate r (used in some possibly empty context program C) admits a recursive logic program. It works as follows. After *selecting* (a suitable renaming for r of) a schema S (with recursive open template T), its open predicates $\underline{c}$ and $\underline{d}$ are somehow *closed* by means of a closed program $\underline{CD}$ (say through reuse of suitable programs stored in repositories), and the current hypothesis program V is set to the union of the context program C, the template T, and the program $\underline{CD}$. Note that V is still open, because predicates p and q remain open. This openness prevents V from covering any element of the clausal evidence set E_r, but the failed proof attempts, for each such piece of evidence, of that coverage may be used to *abduce* clausal evidence sets E_p and E_q for p and q, respectively, using the oracle O_r if necessary. Non-recursive closed programs P_p and P_q for p and q are then *induced* from E_p and E_q, respectively, which induction thus cannot be done through a recursive call to sbis. If the program P_q is considered *acceptable* according to some heuristic criterion, then the closed output program P_r is assembled by closing the open program V with the programs P_p and P_q. Otherwise, the algorithm sbis conjectures that q had better be implemented by a recursive program (i.e., it conjectures necessary/useful predicate invention), and it should of course call itself for this task. The parameters of that self-call are technique-dependent (see below for sample approaches). However, the parameters of the invented predicate are dictated by the template of the chosen schema. The closed output program P_r is assembled either through the recursive call to sbis itself (using the context program facility) or through some operations after that call to sbis.

Instantiating this generic algorithm (which is itself an open program) can be done by providing clauses for its open predicates. This is done differently by the various techniques, as shown next.

Historical Flashback. Many schema-biased synthesisers result from a more or less direct transposition and extension to logic (or rather: relational) programming of the best inductive synthesisers of recursive functional programs, namely the pioneering THESYS [75] and its subsequent generalisation, called *BMWk* [44] [46]. Detailed surveys of the field of inductive synthesis of functional programs exist [8] [28] [70]. There seems to have been some disillusion in that community in the late 1970s, witness the dearth of papers published ever since.

In the early 1980s, MIS [68] [69] (see Section 3.3) and other pioneering techniques of the logic programming and machine learning communities brought a new elan, due to a more powerful theory (logic and relational programming) and a wealth of new ideas through their cross-fertilisation, eventually giving rise to a new branch of artificial intelligence called Inductive Logic Programming (ILP). The added value was in the concepts of background knowledge and declarative bias, in extended evidence languages, in more powerful induction operators, in the inducability of programs for semantic manipulation relations, and in the inducability of both non-recursive and recursive logic programs with the same technique. Curiously, program schemas were a lost value, and were only “rediscovered” in the late 1980s.

Recently, there was a correction and even further generalisation of *BMWk* resulting from a reformulation and formalisation in a term rewriting framework [50]. However, this proposal has not been further pursued (yet), and it still features many of the drawbacks of the original technique, namely absence (and hence no use) of background knowledge, inability to perform necessary/useful predicate invention,[3] and inability to induce programs for semantic manipulation relations.

Similarly, there also was a reformulation and formalisation of *BMWk* in the simply-typed λ-calculus with higher-order unification [35] [36]. However, it also inherits the disadvantages of the original technique.

CRUSTACEAN and CILP. The first two techniques are closely related and were designed by overlapping teams.

The evidence language of the **CRUSTACEAN** technique [1] [2] is ground literals (positive and negative examples), and the evidence may be arbitrarily chosen. No bias is used, other than that the hypothesis language is definite programs of the following template:

$$r(\ldots) \leftarrow$$
$$r(\ldots) \leftarrow r(\ldots)$$

Synthesis is data-driven and passive. There is no usage of background knowledge and no possibility of any kind of predicate invention because of the template. The technique can handle only one relation at a time, and it must be a syntactic manipulation relation. The assumption is thus that a program to be induced consists of one unit base clause *B* and one purely recursive clause *R* (containing only predicate symbol r).

3. It actually tries to *avoid* necessary predicate invention, namely by transformation of the evidence through generalisation (accumulator introduction). However, this avoidance method is not guaranteed to always be successful [29].

The technique does not fit the generic algorithm. It first makes a structural analysis (see [2]) of the positive examples, based on the following observation: a positive example P_i can be proven by resolving B_i and R repeatedly, where B_i is an instance of B. Therefore, the parameters of B_i are sub-terms of the parameters of P_i. For instance, for P_i = last(a,[c,a]), B_i = last(a,[a]), and R = last(A,[B,C|T]) ← last(A,[C|T]), this is the case (where last(E,L) holds iff term E is the last element of list L). As a result of this analysis, the technique produces annotations of the positive examples. These annotations are then used to find B and R, as follows.

The base clause B is computed by taking the lgθ of a set of candidates B_i that are extracted from these annotations. If the lgθ of one such set is an over-general base clause (i.e., which covers negative examples), then backtracking occurs to an alternative set. For instance, an inadequate set extracted from the annotations of the positive examples last(a,[c,a]) and last(b,[e,d,b]) is {last(a,c), last(b,d)}, because its lgθ would yield the base clause last(A,B) ←, which covers all negative examples (not listed here). An adequate set is {last(a,[a]), last(b,[b])}, as its lgθ yields the base clause last(A,[A]) ← , which covers no negative examples.

The recursive clause R is computed in the following way. Its head is the lgθ of all the *iterative decompositions* (see [2]) of all the examples from which the chosen adequate set was obtained. For instance, the iterative decompositions of the two positive examples above are:

```
last(a,[c,a])                              last(b,[e,d,b])
                                           last(b,[d,b])
```

The lgθ of these atoms is last (A,[B,C|T]), and is taken as the head of the recursive clause. The recursive call is constructed by again using the annotations obtained by the structural analysis (see [2]). This here yields last(A,[C|T]), so that the final program is:

```
last(A,[A]) ←
last(A,[B,C|T]) ← last(A,[C|T])
```

When the schema is inadequate, the technique requires that the specifier already has an idea of what a possible program could look like. For instance, a positive example of reverse(L,R) may then be given as follows:

```
reverse([1,2],append(append([],[2]),[1]))
```

where the specifier hardwires an idea of how to revert a list by using append (note that the idea is *represented* by a *given* functor named append, rather than *computed* by means of an *invented* predicate).

The technique does not need to be given any examples covered by the base clause, as it constructs its own such examples from those covered by the recursive clause. The technique is a generalisation of the LOPSTER technique [47], which requires the positive examples to be on the same resolution path, and hence carefully crafted. However, the latter can also induce programs with left-recursive clauses and is thus biased by a more general template.

The **CILP** (*Constructive Inductive Logic Programming*) technique [48] interactively induces programs for syntactic manipulation relations, and makes no usage of background knowledge. The technique fits the generic algorithm as follows:

- the evidence E_r is ground literals (positive and negative examples), and may be arbitrarily chosen;
- the oracle O_r is the specifier;
- the context program C is the empty program ■;
- selectSchema always amounts to selecting the following (divide-and-conquer) schema:

  ```
  r(…) ←
  r(…) ← r(…), q(…)
  ```

  hence featuring a priori no $\underline{c}$, $\underline{d}$, p predicates; this means that most computations are performed through unification and the recursive call, which may be considered as if there were solve (as p) and decompose (as $\underline{d}$) open predicates in the template, as in the template of Example 1, but with the restriction that their instances must be implemented in terms of equality atoms only, so that they can be unfolded so as to yield a program of the template above; the technique is thus schema-based;
- close-$\underline{cd}$, abduce, and induce are intricately merged into one step, as briefly illustrated here by means of the length predicate, where length(L,N) holds iff integer N is the length of the list L:
  - the *recursive clause* is first computed, by a method called *sub-unification* (see [47]), which is based on the structural differences of the parameters of a pair of selected positive examples, and which determines a recursive clause that inverts the largest number of resolution steps between these examples; from the examples

length([a,b],s^2(0)) and length([a,b,c,d],s^4(0)), it is found to be length([H|T],s(N)) ← length(T,N); an alternative clause, but that inverts fewer resolution steps and covers fewer test examples, is length([G,H|T],s^2(N)) ← length(T,N); this is a remarkable feature, since the technique can thus work from fewer examples, which is especially useful when performing necessary predicate invention, as the abduced examples of the invented predicate are sometimes sparsely distributed over its intended relation;
  – the *base clause* is then computed using the following observation: the base clause is a unit clause used by a recursive program in the last step of a refutation; it is thus found by taking the lgθ of the unresolved facts; for instance, the first example can be resolved twice (using the recursive clause) to obtain the unresolvable fact length([],0), whereas resolving the second example four times yields again the same fact length([],0). The base clause is thus length([],0) ← ; the technique thus does not need to be given any examples covered by the base clause, as it constructs its own such examples from those covered by the recursive clause;
  – the parameters of q initially are all the variables of the recursive clause; then, *harmful* variables are heuristically eliminated (see [48]) and *useful* variables are kept (their elimination by projection would provoke the abduction of the same example of q from a positive and a negative example of r, hence making it undecidable whether that example of q is positive or negative); such abduction is made by SLD execution, using the oracle if necessary;
- acceptable conjectures necessary predicate invention if every program induced for every selected pair of positive examples covers some negative example;
- recurse amounts to the following conjunction: sbis(E_q,■,■,Q), P_r = V ∪ P_p ∪ Q; for the self-call, there is thus no oracle, and no use is again made of the context program facility: the new synthesis is thus independent of the old one, and their resulting programs are then joined; note that the new evidence is less numerous than the old one, so that there is a limitation to how many predicate inventions can be performed in a chain; fortunately, the technique handles sparse evidence sets quite well.

FORCE2. The evidence for the FORCE2 technique [18] may consist of arbitrarily chosen ground literals (positive and negative examples). The technique also requires a *depth complexity* of the program to be induced, and a function for determining whether an atom is an (instance of the) base case of the recursion. For instance, for the append predicate, the source could give the following:

maxdepth(append(X,Y,Z)) = *length*(X) + 1
basecase(append(X,Y,Z)) = if X=[] then *true* else *false*

The source need only supply an upper bound on the depth complexity (not a precise bound), and a sufficient (not both necessary and sufficient) condition for membership in the base case.

The hypothesis language is two-clause linear and closed recursive *ij*-determinate definite programs. A clause is *linear and closed recursive* if its body has a single recursive atom with no output variables. Thus, the template is:

r(…) ← $\underline{c}$(…)
r(…) ← $\underline{d}$(…), r(…)

where each atom in $\underline{c}$ and $\underline{d}$ is an *ij*-determinate literal that is defined in the background knowledge, and the recursive atom r(…) has no output variables. The technique can handle only one (syntactic or semantic manipulation) relation at a time and cannot do any kind of predicate invention. It requires background knowledge that includes only predicates of arity *j* or less, and of a depth bound *i*. The technique is passive, data-driven, but not fully implemented. The identification criterion is PAC-identification.

The technique first splits the positive examples into two subsets by using the *basecase* function, thus obtaining examples of the base clause and of the recursive clause. Then, the rlgθs *B* and *R* of these two example sets relative to the background knowledge are used as initial guesses for the base clause and recursive clause, respectively. For instance, the rlgθs of the positive examples append([],[1],[1]), append([],[2,3],[2,3]), append([1],[],[1]), and append([1,2],[3],[1,2,3]) are:

append(X,Y,Z) ← Y=[W|V], X=[], Z=Y (*B*)
append(X,Y,Z) ← X=[W|V], Z=[T|U], W=T (*R*)

Next, for each possible recursive atom *L* over the variables in *R*, the technique proceeds to the following simultaneous refinement of the *base clause* and the *recursive clause*:

- Suppose the chosen (and adequate) recursive atom is append(V,Y,U). For each positive example *e*, if it is a base case (which is determined using the *basecase* function), then *B* is replaced with its lgθ with *e*; otherwise, *R* is replaced with its lgθ with *e*. For instance, for *e* = append([1,2],[3],[1,2,3]), it is found that *e* is not a base case, therefore *R* is generalised such that it covers *e*. Here, *R* remains unchanged, because it already covers *e*. Next, the corresponding instance *i* of the recursive atom is computed; if *i* is a base case, then *B* is replaced with its lgθ with *i*; otherwise, *R* is replaced with its lgθ with *i*. Here, the corresponding instance of append(V,Y,U) is append([2],[3],[2,3]), which is not a base case, so *R* is replaced with its lgθ with append([2],[3],[2,3]), which again does not change *R*. This instantiation process continues until a base case is obtained. Here, the atom append([ ],[ ],[ ]) is now obtained, and determined to be a base case. So, *B* is generalised to its lgθ with append([ ],[ ],[ ]), yielding: append(A,B,C) ← A=[], C=B. After doing this with all positive examples, the chosen recursive atom is added to the end of *R* to obtain the recursive clause of the final program. Next, it is checked whether the program covers any negative examples. If it covers some, then it is rejected and another possible recursive atom is tried.

- Now, suppose that the recursive atom was chosen incorrectly: how can this be detected? For instance, let *L* be append(X,X,Z). Then, for the example append([1,2],[3],[1,2,3]), the same calls would be generated repeatedly. This is detected by means of the *maxdepth* function when the depth bound is exceeded, and an error is signalled to indicate that there is no valid generalisation of the program that covers the example. For incorrect recursive atoms that do not provoke looping beyond the depth-bound, the synthesis might end up with an over-general hypothesis. However, this can be detected by using sufficient negative examples.

Note that there are polynomially many possible recursive atoms to be tested.

SIERES. The SIERES technique [79] is passive, data-driven, and can handle one (syntactic or semantic manipulation) relation at a time. Mode declarations are used as search bias. The hypothesis language is definite programs. The background knowledge consists of definite clauses. The technique makes use of schemas called *argument dependency graphs* (ADG), which specify the number of literals within a clause and the argument dependencies between them. For instance, such a graph is r([H|T],R) ← r(T,Q), q(H,Q,R). A literal L_1 *depends on* a literal L_2 if they share a variable that is an output variable of L_1 and an input variable of L_2 (as indicated in the mode declarations). The technique fits the generic algorithm as follows:

- the evidence E_r is ground literals (positive and negative examples), which may be arbitrarily chosen;
- there is no oracle O_r;
- the context program C is the empty program ■;
- selectSchema selects an ADG from a set of ADGs; the technique is thus schema-guided;
- close-<u>cd</u>, abduce, and induce are intricately merged into one step:
  - there is no indication how the *base clause* is discovered, but it seems done before finding the recursive clause;
  - the *recursive clause* is computed as follows: the technique first computes the lgθ of the positive examples, and uses it as the head for the recursive clause; if this lgθ is over-general (i.e., if it covers any negative examples), then it is specialised, using the mode declarations and the selected ADG; the parameters of possible body literals (using predicates from the background knowledge or the top-level predicate) are restricted to *critical terms* (unused input and unused output terms); new variables and/or uncritical terms are used as parameters only when there are more parameters of the predicate than critical terms;
- acceptable conjectures necessary predicate invention if none of the existing predicates yields a correct specialisation of the recursive clause; the parameters of the invented predicate are selected so that the resulting clause contains no more critical terms;
- recurse amounts to the following conjunction: sbis(E_q,■,■,Q), $P_r = V \cup P_p \cup Q$; for the self-call, there is thus again no oracle and no context program: the new synthesis is thus independent of the old one, and their resulting programs are then joined; note that the new evidence is less numerous than the old one, so that there is a limitation to how many predicate inventions can be performed in a chain; unfortunately, the technique cannot handle the sparseness problem.

Example 9: Given the positive examples sort([1],[1]), sort([3,1],[1,3]), and sort([2,4,1],[1,2,4]), suppose the over-general clause induced (using the mode declarations and the ADG given above) is sort([H|T],S) ← sort(T,Y). Let the background knowledge include only a program for the ≤ predicate. Then, none of the existing predicates yields a correct specialisation of the clause conforming to the ADG. This initiates necessary predicate invention. The critical terms of the over-general clause are H, Y, S. Thus, the new literal is q(H,Y,S), and the abduced positive examples are q(1,[],[1]), q(3,[1],[1,3]), and q(2,[1,4],[1,2,4]). This denotes an example set of the insertion of a number into a sorted list of numbers. For space reasons, we omit the details of the self-call. ♦

TIM. The evidence for TIM (*The Induction Machine*) [40] may consist of arbitrarily chosen ground atoms (positive examples). The hypothesis language is definite programs that fit a template with exactly one base clause and one recursive clause, where the unique recursive call is the last body atom. The background knowledge is composed of definite clauses. Mode declarations are provided as a search bias. The technique can handle only one (syntactic or semantic manipulation) relation at a time.

The basic idea is to construct explanations of the examples in terms of the background knowledge, and then to analyse these explanations in order to induce a program. The technique first computes saturations of the examples. A clause F is a *saturation* of an example E relative to background knowledge B if F is the most specific reformulation (under implication) of E relative to B. A clause F is a *reformulation* of a clause E relative to background knowledge B if $B \wedge F \equiv B \wedge E$. For instance, for the examples E_1, E_2, the mode declarations M_1, M_2, and the background knowledge clauses B_1, B_2 below:

E_1: member(b,[a,b])	E_2: member(e,[c,d,e,f])
M_1: dec(+,–,–)	M_2: equal(+,+)
B_1: dec([H|T],H,T) ←	B_2: equal(X,X) ←

the following clauses F_1 and F_2 are the saturations of E_1 and E_2 relative to B:

F_1: member(b,[a,b]) ← dec([a,b],a,[b]), dec([b],b,[]), equal(b,b)
F_2: member(e,[c,d,e,f]) ← dec([c,d,e,f],c,[d,e,f]), dec([d,e,f],d,[e,f]), dec([e,f],e,[f]), equal(e,e)

First, the *recursive clause* is computed as follows. The technique analyses (see [40]) the saturations so as to find common *structural regularities* in pairs of saturations. On finding such pairs, the technique adds a ground recursive atom (suggested by the analysis) to the end of the body of each saturation. The recursive clause is then the lgθ of these augmented saturations.

Then, the *base clause* is constructed as follows. Instances of the head of the base clause are computed by again exploiting the structural regularity information found in the saturations computed earlier. The base clause is then the lgθ of the saturations of these instances.

For our problem, the technique uses the saturations F_1 and F_2 to infer the following program:

```
member(X,Y) ← dec(Y,X,Z)
member(X,Y) ← dec(Y,Z,W), member(X,W)
```

The technique is passive, and is not able to perform any kind of predicate invention.

SYNAPSE, DIALOGS, and METAINDUCE. The next three techniques are very similar to each other, so that we can discuss them together. They all target programming assistance applications, so that the source is a specifier.

The **SYNAPSE** (*SYNthesis of Algorithms from PropertieS and Examples*) technique [28] [31] is based on a divide-and-conquer schema that subsumes the one of Example 1, in the sense that the arity of r and the number of recursive calls are parameterised, hence providing more flexibility. Also, the primitive and nonPrimitive checks are each divided into a syntactic check (called minimal and nonMinimal, respectively) and a semantic check (called discriminate). Multiple base clauses and multiple recursive clauses are possible, through multiple clauses for solve and compose, respectively.

The evidence language is (non-recursive) Horn clauses describing a single intended relation. Ground unit clauses are called (positive) *examples* and data-drive the synthesis; all other clauses are called *properties* and are used to find the instances of solve, compose, and discriminate. No other bias is given, though types are inferred from the examples. Mode and determinism information are not required, because the focus is on synthesising the logic component of logic programs.

Here is some evidence for delOdds(L,R), which holds iff R is integer-list L without its odd elements:

delOdds([],[]) ←	
delOdds([1],[]) ←	delOdds([A],[]) ← odd(A)
delOdds([2],[2]) ←	delOdds([B],[B]) ← ¬odd(B)
delOdds([3,4],[4]) ←	
delOdds([6,8],[6,8]) ←	

The rationale behind properties becomes apparent now: since examples alone cannot express everything the specifier *must* know about delOdds, namely the odd concept, a way had to be found to overcome this limitation. Properties thus allow the synthesis of programs for semantic manipulation relations without a background knowledge usage miracle (see Section 4.2.3). Nothing prevents giving too complete properties, such as a correct recursive program, but the technique works from as little evidence as given above.

The hypothesis language is normal logic programs, where negation is restricted to the discriminants and appears there by extraction from the properties (i.e., it can only be applied to primitive predicates and could thus be avoided by using the complementary primitives in the properties).

Synthesis is passive, although there is an *expert* mode where the technique asks for a preference among the possible instances of the minimal, nonMinimal, and decompose place-holders, rather than non-deterministically choosing each from a repository. These problem-independent repositories form the (partitioned) background knowledge. The technique fits the generic algorithm as follows:

- the evidence E_r is partitioned into an example set Ex_r and a property set $Prop_r$, as described above;
- the oracle O_r always is E_r;
- the context program C always is the empty program ■;
- selectSchema always amounts to selecting the abovementioned divide-and-conquer schema, but without discriminate (see below), and taking solve as p and compose as q; the technique is thus schema-based;
- close-cd closes the minimal, nonMinimal, and decompose predicates by reuse of suitable programs from the repositories;
- abduce computes example (not evidence) sets of p and q as follows:

$$Ex_p \cup Ex_q = \{e' \mid \exists\, e \in Ex_r \,.\, (V \cup \{e'\}) \oplus (O_r \setminus \{e\}) \vdash_{SLDNF} e\}$$

  where $T \oplus S \vdash_{SLDNF} G$ denotes that, for proving goal G with SLDNF resolution, the theory T is used for all predicates defined in T, except for those predicates that are also defined in S, for which theory S is used instead; note that the currently proved example e has to be deleted from the oracle, because otherwise there would be a trivial proof;
- induce works *separately* on p and q; for each of them, it divides the example set into maximal subsets such that their lgθs are not too general according to what is called a *construction mode* [27] (see Example 10 below); this division is not necessarily a partition and can be performed by a clique cover algorithm [27]; each of these lgθs constitutes one (unit) clause of P_p or P_q; note that there can thus be several clauses for p and q, so that there can essentially be, after unfolding of p and q, several base clauses and several recursive clauses for r;
- acceptable conjectures necessary or useful predicate invention if P_q has more clauses than there are properties in $Prop_r$, because otherwise there would not have been any compression in the number of clauses between the evidence and the program; this obviously requires the properties to be carefully crafted;
- recurse amounts to the following conjunction: $sbis(E_q, E_q, ■, Q)$, $P_r = V \cup P_p \cup Q$; for the self-call, the oracle is thus again the provided evidence, and no use is again made of the context program facility: the new synthesis is thus independent of the old one, and their resulting programs are then joined; note that the new evidence is less numerous than the old one, so that there is a limitation to how many predicate inventions can be performed in a chain; nothing is foreseen to detect and handle the sparseness problem, if it occurs;
- all calls to sbis must be followed by a call to an addDisc step, which adds calls to discriminate and abduces clauses for it, using $Prop_r$ (see [28]).

Let us now analyse the behaviour of this instance of the generic algorithm on an example.

Example 10: For delOdds, suppose close-cd produces the intermediate program (after some unfolding):

```
delOdds(L,R) ← L=[], solve(L,R)
delOdds(L,R) ← L=[HL|TL], delOdds(TL,TR), compose(HL,TR,R)
```

Then, abduce finds the following examples of solve and compose from the examples of delOdds:

```
solve([],[]) ←
                              compose(1,[],[]) ←
                              compose(2,[],[2]) ←
                              compose(3,[4],[4]) ←
                              compose(6,[8],[6,8]) ←
```

Next, induce works as follows. The lgθ of all the examples of compose would be compose(P,Q,R) ← , which is over-general according to the construction mode for compose. The latter is a problem-independent pre-computed constraint stating that the third parameter *must* somehow be constructed using the second parameter and *may* possibly even be constructed using the first parameter. To satisfy this mode, the evidence is divided into subsets: the first and third clauses have compose(H,T,T) ← as lgθ, whereas the second and fourth clauses have compose(H,T,[H|T]) ← as lgθ. These lgθs satisfy the mode, and there is no such division into less than two subsets. The lgθ of the evidence of solve is its only element, which satisfies the construction mode for solve.

Since there are two clauses now for compose and two properties for delOdds, the acceptable heuristic accepts this result, and the program now is assembled as follows (after some unfolding):

```
delOdds(L,R) ← L=[], R=[]
delOdds(L,R) ← L=[HL|TL], delOdds(TL,TR), R=TR
delOdds(L,R) ← L=[HL|TL], delOdds(TL,TR), R=[HL|TR]
```

The discriminants are then abduced by addDisc, and the final program is:

```
delOdds(L,R) ← L=[], R=[]
delOdds(L,R) ← L=[HL|TL], odd(HL), delOdds(TL,TR), R=TR
delOdds(L,R) ← L=[HL|TL], ¬odd(HL), delOdds(TL,TR), R=[HL|TR]
```

This program is correct w.r.t. the intended relation, hence has a 100% accuracy against any test set. ♦

Another sample run, featuring necessary or useful predicate invention, is not given here. We will illustrate this by showing it on the successor technique DIALOGS (see below). The overall technique is much more powerful than explained here, a lot of its additional sophistication going into the detection and handling of constant parameters (which do not change through recursive calls, such as variable Z in the template template (*sic*) or integer E in insert, see Example 6) and of useless recursive calls (when the induction parameter is only traversed partially, such as L in insert).

Unfortunately, for time reasons, the technique was never implemented up to its full power, as it is described in [28]. However, insights gained during its design and experimentation led to the design of the DIALOGS technique, described hereafter. But let us first discuss METAINDUCE, because it is very close to SYNAPSE.

The **METAINDUCE** technique [37] is almost exactly a particular case of SYNAPSE. The main contribution is an extremely elegant implementation based on a meta-programming approach, which is a big step towards actual schema guidance. The hardwired schema is an instance of the SYNAPSE one, namely for ternary relations, with the induction parameter of type list, with exactly one base clause (when the list is empty), with exactly one recursive clause (when the list is non-empty), with head-tail decomposition of the list (i.e., exactly one recursive call), and without discriminants. The evidence language is positive and negative examples, and there is no background knowledge nor properties, hence a restriction to syntactic manipulation relations. The hypothesis language is definite logic programs. The close-$\underline{cd}$ and addDisc steps do nothing (as $\underline{c}$ and $\underline{d}$ are empty and as there are no discriminants), and SLD execution suffices for abduce. There is no oracle, so the abduced evidence of q is initially made of conjunctions of q atoms that share some variables (see [37] for details on how to derive actual examples from such evidence). The induce step computes the lgθs of the *entire* example sets, due to the absence of construction modes, and thus produces a unique (unit) clause, which may be over-general (as seen in Example 10). The acceptable heuristic conjectures necessary predicate invention upon coverage of some abduced negative q example by the q clause. The technique and its implementation are only considered proof-of-concept prototypes by their designers.

The **DIALOGS** (*Dialogue-based Inductive/Abductive LOGic program Synthesiser*) technique [30] [80] results from an effort at building a fully interactive version of SYNAPSE, and at extending its power while at the same time simplifying its machinery. The main objective was to take all burden from the specifier by having the technique ask for exactly and only the information it needs. As a result, no evidence needs to be prepared in advance, as the technique invents its own evidence and queries the specifier about it. This is suitable for all levels of expertise of human users, as the queries are formulated in the specifier's (initially unknown) conceptual language, in a program-independent way, and such that the specifier *must* know the answers if s/he really feels the need for the program. The technique is schema-guided, and currently has two schemas (divide-and-conquer and accumulate). The evidence language implicitly amounts to (non-recursive) normal programs. Type declarations are available as language bias. The hypothesis language is recursive normal programs with possibly multiple base clauses and recursive clauses. The technique fits the generic algorithm as follows:

- the evidence E_r is empty;
- the oracle O_r is the specifier;
- the context program C is arbitrary;
- the predicate sequence $\underline{c}$ is empty in the template template (*sic*);
- selectSchema interactively selects a schema and a *strategy*, the latter stating which open predicates play the roles of $\underline{d}$, p, and q, respectively; the technique is thus really schema-guided;
- close-$\underline{cd}$ interactively closes the $\underline{d}$ predicates by reuse of suitable programs from the repositories;
- abduce computes evidence sets of p and q as follows:

  $$E_p \cup E_q = \{e' \mid \exists\, g \in G_r\,.\,(V \cup \{e'\}) \oplus O_r \vdash_{SLDNF++} g\}$$

  where G_r is a finite set of most-general goals for r, obtained by varying the size of the induction parameter; also, SLDNF++ denotes *extended SLDNF execution* [45], which can also prove certain clausal goals; the questions to the oracle about p and q are reverse-engineered into questions about r; all answers by the oracle are stored as *judgments*, to prevent asking the same question twice;
- induce works *simultaneously* on p and q; for each pair of corresponding evidence clauses of p and q, it deletes one (see [80]) and divides the remaining clauses of each set into maximal subsets such that their lgθs are not too general according to a construction mode; each of these lgθs constitutes one clause of P_p or P_q;
- acceptable conjectures necessary or useful predicate invention if P_q is empty, because otherwise all the computations would be non-recursively performed through the base clauses;
- recurse amounts to sbis(■,O_r,V ∪ P_p,P_r); thus, the evidence is again empty, but the context program facility is used here: the new synthesis is *not* independent of the old one, in the sense that abduction of evidence will again start from goals for r; the questions about p and q (of the old q thus) being reverse-engineered into questions about r, the oracle for r can be used here as well; the judgments stored at the previous level avoid duplicate questions; the new evidence is thus *not* necessarily less numerous than the old one, and some heuristic (see [80]) detects and handles the sparseness problem, if it occurs.

Let us now analyse the behaviour of this instance of the generic algorithm on an example, featuring necessary/useful predicate invention.

Example 11: Consider the following template of a schema called divide-and-conquer:

$r(X,Y,Z) \leftarrow solve(X,Y,Z)$
$r(X,Y,Z) \leftarrow decompose(X,Z,H,X_1,\ldots,X_t),\ r(X_1,Y_1,Z),\ \ldots,\ r(X_t,Y_t,Z),\ compose(H,Z,Y_1,\ldots,Y_t,Y)$

Consider the strategy dc_1 for this schema, expressing that $\underline{c}$ is empty, $\underline{d}$ is decompose, p is solve, and q is compose. Another strategy is dc_2, which is dc_1 where the roles of decompose and compose are exchanged.

Here is a sample transcript of a dialogue for the reverse(L,R) relation, which holds iff R is the reverse of list L. Proposed answers (if any) are between curly braces "{…}" and can thus be chosen as defaults by computationally naive specifiers; the specifier's actual answers are in *italics* (and are printed even when the default was chosen); the comma "," stands for conjunction; and the semi-colon ";" stands for disjunction. The dialogue is fully backtrackable, which yields the opportunity to synthesise several programs.

```
Predicate declaration?   reverse(L:list(term),R:list(term))
Schema? {divide-and-conquer, accumulate}   divide-and-conquer
Strategy? {dc1, dc2}   dc1
Induction parameter? {L}   L
Result parameter? {R}   R
Decomposition? {L=[HL|TL]}   L=[HL|TL]
When does reverse([],R) hold?   R=[]
When does reverse([A],R) hold?   R=[A]
When does reverse([A,B],R) hold?   R=[B,A]
When does reverse([A,B,C],R) hold?   R=[C,B,A]
When does reverse([A,B,C,D],R) hold?   stop it!
```

Note how all questions are about reverse, and that the specifier decided (at her/his own risk) when s/he had given enough information. However, it *is* enough for inferring the following program (after some unfolding):

```
reverse(L,R) ← L=[], R=[]
reverse(L,R) ← L=[HL|TL], reverse(TL,TR), compose(HL,TR,R)
```

```
compose(E,L,R) ← L=[], R=[E]
compose(E,L,R) ← L=[HL|TL], compose(E,TL,TR), R=[HL|TR]
```

Let us inspect in some detail what happened. The abduce step produces the following pairs of corresponding pieces of evidence of solve and compose:

solve([],[]) ←	(*none*)
solve([A],[A]) ←	compose(A,[],[A]) ←
solve([A,B],[B,A]) ←	compose(A,[B],[B,A]) ←
solve([A,B,C],[C,B,A]) ←	compose(A,[C,B],[C,B,A]) ←

The induce step produces zero clauses for compose (after deleting all its evidence), and three clauses for solve. The acceptable heuristic thus conjectures necessary/useful predicate invention, so that DIALOGS calls itself with as (open) context program the first two clauses of the program above. Before that, it constructs the predicate declaration compose(HL:term,TR:list(term),R:list(term)) and switches its execution mode from (the initial) *aloud* to *mute*, meaning that all constructed default answers will be automatically selected. Indeed, the specifier has no idea what compose means (especially that the intended relation of compose depends on the one of decompose, and is thus not unique), and should thus not have to answer queries about it. DIALOGS also constructs *hints* for the default answer construction, stating here that the divide-and-conquer schema with strategy dc_1 ought to be selected, that TR ought to be the induction parameter, that R ought to be the result parameter, and that HL ought to be a passive parameter. With this setup, plus the judgments stored during the old synthesis, the new synthesis can be run entirely off-line (as witnessed by the transcript above) and yet collect evidence of the solve and compose predicates of (the old) compose. More evidence can be gathered for these predicates, so the specifier is actually again asked what the reverse of a four-element-list is, because there is no judgment for it. Suppose s/he still believes s/he already said everything useful about reverse. This time, from the results of the abduce step, the induce step produces clauses that do not lead to a conjecture of necessary/useful predicate invention by the acceptable heuristic, and the final program is as above. ♦

3.2 Schema-less Synthesis

SPECTRE II and MERLIN. The following two theory-guided techniques are not really related, but we grouped them together because they were designed by the same person. An initial theory cannot (really) be seen as a schema, as it is usually modified during the induction, and thus does not (really) guide the induction process. Furthermore, an initial theory is problem-specific, whereas a schema would have to be problem-independent.

The inputs to the **SPECTRE** II (*SPECialisation by TRansformation and Elimination*) technique [9] are carefully crafted ground literals (positive and negative examples) as evidence of possibly multiple (syntactic or semantic manipulation) relations, and an overly general initial theory that is already recursive. The hypothesis language is definite programs. There is no background knowledge, nor any kind of bias. The technique cannot perform any kind of predicate invention.

The technique only works under the following assumptions: all positive examples are logical consequences of the initial theory, there is a finite number of refutations of the positive and negative examples, and there are no positive and negative examples that have the same sequence of input clauses in their refutations.

The technique works as follows. First, as long as there is a refutation of a negative example such that all clauses used in this refutation also appear in all refutations of the positive examples, an atom in a clause of the current program is unfolded. Second, for each refutation of a negative example, an input clause that is not used in any refutation of any positive example is removed. The clauses to be unfolded or removed can be selected randomly; this does not affect the correctness of the induced program w.r.t. the training set.

Example 12: Suppose the positive examples odd(s(0)), odd(s^3(0)), odd(s^5(0)), the negative examples odd(0), odd(s^2(0)), odd(s^4(0)), and the following initial theory are given:

odd(0) ←	(c_1)
odd(s(X)) ← odd(X)	(c_2)

According to the first step, there is a negative example, namely odd(0), for which all clauses, namely c_1, in its refutation appear in all refutations of the positive examples. If one selects c_2 and unfolds the atom in its body, then the following new program is obtained:

```
odd(0) ←                          (c1)
odd(s(0)) ←                       (c3)
odd(s2(X)) ← odd(X)               (c4)
```

No other unfolding need now be done. Then, according to the second step, clause c_1 must be removed. This results in the following new program:

```
odd(s(0)) ←                       (c3)
odd(s2(X)) ← odd(X)               (c4)
```

which is correct. ♦

The correctness of the technique is proved by a theorem [8]. The technique is passive, and actually uses heuristics during clause selection for unfolding and removing. The technique is a generalisation of the **SPECTRE** technique [11], in the sense that it no longer requires the examples to be of the same relation.

The inputs to the **MERLIN** (*Model Extraction by Regular Language INference*) technique [10] are carefully crafted ground literals (positive and negative examples) as evidence of one (syntactic or semantic manipulation) relation, and an overly general initial theory that is already recursive. The hypothesis language is definite programs. The technique is passive and resolution-based. There is no background knowledge, nor any kind of bias.

Previous resolution-based approaches to theory-guided induction produce hypotheses as sets of resolvents of the initial theory, where allowed sequences of resolution steps are represented by resolvents. However, this is not always possible. For instance, suppose the positive examples p([a,b]), p([a,a,b,b,b]), the negative examples p([b,a]), p([a,b,a]), and the following initial theory are given:

```
p([]) ←                           (c1)
p([a|L]) ← p(L)                   (c2)
p([b|L]) ← p(L)                   (c3)
```

One can then find the following characterisation of the sequences of resolution steps that are used in the refutations of the positive examples, where the characterisation does not hold for the refutations of the negative examples: first the clause c_2 is used an arbitrary number of times, then the clause c_3 is used an arbitrary number of times, and finally c_1. This cannot be expressed by a set of resolvents of the given theory, but rather by the following theory:

```
p([]) ←                           (c1)
p([a|L]) ← p(L)                   (c2)
p([b|L]) ← q(L)                   (c4)
q([]) ←                           (c5)
q([b|L]) ← q(L)                   (c6)
```

Note that predicate q had to be necessarily invented.

The technique has a new approach to solving this representation problem. It tries to induce a finite-state machine that represents allowed sequences of resolution steps. It thus views refutations of positive (resp. negative) examples as strings in (resp. not in) a formal language, and represents this as a finite-state machine, where the final states correspond to either a positive example or a negative example. Then, this automaton is reduced by merging the start states, and is made deterministic. Next, the set of sequences allowed by the given initial theory is represented as a context-free grammar, and then a new context-free grammar is derived that represents the intersection of the former grammar and the automaton. Finally, this new grammar is used to produce the final program, possibly with predicate invention. The technique assumes that all positive examples are logical consequences of the initial theory, and that there are no positive and negative examples that have the same sequence of input clauses in their refutations. Describing this in detail is beyond the scope of this paper, and we refer to the original article [10]. Suffice it to say that, from the initial theory and examples above, the technique infers the correct specialisation above.

SMART. The evidence for the SMART [54] technique may consist of arbitrarily chosen ground literals (positive and negative examples) for one (syntactic or semantic manipulation) relation. The positive examples may lie on non-intersecting resolution paths of induced programs, and may thus be quite sparse. The hypothesis language is definite programs. The background knowledge is definite clauses with mode and type information, and a search bias is given in the form of an upper bound on the length of clauses.

The unique (unit) *base clause* is first induced, in a way similar to CRUSTACEAN (see Section 3.1), the main difference being that the technique can generate its own negative examples for the base clause, rather than using only the user-supplied ones.

The *recursive clause* is next induced, as follows. Candidate recursive clauses from the hypothesis space are enumerated top-down in an exhaustive manner, and one such clause is selected if it is correct w.r.t. the set of positive and negative examples. Search explosion is controlled by enforcing the given upper bound on clause lengths and by disallowing inactive variables. Let us illustrate this on an example.

Example 13: Assume the technique constructs a recursive clause for the sort predicate, yielding the following clause at some moment:

$\mathsf{sort(L,S)} \leftarrow \mathsf{dec(L,H,T),\ part(T,H,L_1,L_2)}$

where dec and part are background predicates. Beyond the first body atom, L has no role to play for the rest of the clause, since it has been replaced by two variables, namely H and T. Similarly, H and T have no roles beyond the second body atom. The technique thus considers variables like L, H, T as *inactive* variables, and it eliminates them from the set {L, S, H, T, L_1, L_2} of possible variables in order to construct the set of *active* variables, which can be used in an atom to be added after the currently last body atom. This restricts the search space. Continuing with the example above, the technique then adds two recursive atoms, yielding the clause:

$\mathsf{sort(L,S)} \leftarrow \mathsf{dec(L,H,T),\ part(T,H,L_1,L_2),\ sort(L_1,S_1),\ sort(L_2,S_2)}$

Note that, during the introduction of a recursive call, the technique does not take into consideration the sort(H,C), sort(T,C), and sort(L,C) atoms, as they would have inactive variables. The clause induced so far does not yield a correct program yet with the base clause, as S still needs to be computed. We omit this here. ♦

However, it is not always possible not to reuse inactive variables since some classes of programs only become inducable if the technique allows reusing inactive variables. This situation is handled using some strategies (see [54]). Also note that the technique cannot perform any kind of predicate invention.

SKILIT. The input of the SKILIT technique [42] may consist of arbitrarily chosen ground literals (positive and negative examples) as evidence of one (syntactic or semantic manipulation) relation, plus mode and type declarations of the involved predicates and (possibly recursive) algorithm sketches as search bias. An *algorithm sketch* [41] [13] is an incomplete representation of the derivation associated with a positive example. An algorithm sketch is represented as a clause $\mathsf{E} \leftarrow \mathsf{L_1,\ldots,L_m}$, where E is a positive example and each L_i is either a ground literal involving a background predicate or a literal of the form $p(…), called a *sketch atom*, involving an undefined *sketch predicate* $p. The body of a sketch represents the derivation related to example E. If there is no sketch for an example $\mathsf{r(t_1,\ldots,t_n)}$, then a *blackbox sketch* is automatically constructed, namely $\mathsf{r(t_1,\ldots,t_n)} \leftarrow \$\mathsf{p(t_1,\ldots,t_n)}$. We do not consider sketches as schemas, because they can be problem-specific whereas schemas ought to be problem-independent. The hypothesis language is definite programs. The background knowledge is a set of definite clauses.

The technique starts with the empty program, and adds one clause (by refinement of an algorithm sketch) at each iteration, provided that clause together with the current program and background knowledge does not cover any negative examples. At the end of each iteration, redundant clauses are removed from the current program (see [42]). This is repeated until two successive programs are the same and all positive examples are covered by the program. *Refinement* of an algorithm sketch is realised by substituting all its sketch predicates by suitable background predicates or by the predicate of the examples (by which way recursion is introduced). If there are no matches between a sketch atom $\$\mathsf{p}_i(\ldots)$ and any of the atoms that are logical consequences of the background knowledge, then that sketch atom is replaced by $\mathsf{b}(\ldots), \$\mathsf{p}_k(\ldots)$, where b is a background predicate that generates the outputs of $\$\mathsf{p}_i$ and $\$\mathsf{p}_k$ is a new sketch predicate. The refined algorithm sketch is then generalised by variablising the parameters of its sketch atoms such that the dataflow is preserved. Note that the technique is passive and cannot perform any kind of predicate invention.

The SKILIT+MONIC technique differs from SKILIT in the way it performs consistency checking. It uses integrity constraints (which are first-order clauses) instead of negative examples. A Monte Carlo method for verifying integrity constraints (MONIC) [43] is used.

Example 14: Suppose given the positive examples sort([],[]) and sort([3,2,1],[1,2,3]), the negative examples sort([3,2],[3,2]) and sort([],[1]), the algorithm sketches $\mathsf{sort([],[])} \leftarrow \$\mathsf{p_1([]),\ \$p_2([])}$ and $\mathsf{sort([3,2,1],[1,2,3])} \leftarrow \mathsf{sort([2,1],[1,2]),\ \$p_3(3,[1,2],[1,2,3])}$, and background knowledge with programs for the insert and null predicates. Synthesis starts by refining the first sketch. Its sketch predicate $\$\mathsf{p_1}$ is determined to be null, since null([]) is a logical consequence of the background knowledge. Its second sketch predicate $\$p_2$ is also found in that way to be null. Synthesis then refines the second sketch. Its sketch predicate $\$p_3$ is found to be insert, since the background knowledge has insert(3,[1,2],[1,2,3]) as a logical consequence. The resulting program is the following:

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

which is a correct program. ♦

3.3 Synthesis by General-Purpose Techniques

There is a wealth of general-purpose techniques that can induce *both* non-recursive *and* recursive logic programs, but they vary greatly in their ability to synthesise recursive programs. Here we just overview some representative techniques that have reasonable behaviour on the latter task, and refer the reader to other overviews [59] [78].

MIS. The introduction of the *Model Inference System* (MIS) [68] [69] is often considered *the* initial breakthrough event of ILP. Although it can identify (in-the-limit) *any* logic program, MIS has mostly been demonstrated through its ability to synthesise recursive programs. The evidence language is ground literals (positive and negative examples) for possibly multiple relations performing any kind of manipulations, and the hypothesis language is definite programs. Additional specification information includes type, mode, and determinism information as search bias, and a list of deemed-to-be-relevant predicates of the background knowledge as language bias (if this basis set includes the predicate(s) for which examples are given, then recursive clauses will be considered by the technique). The background knowledge consists of definite clauses. Induction incrementally proceeds bottom-up, starting from the initial theory *P* (or the empty program, if none is given): [4]

```
repeat
  read the next example
  repeat
    if P is not complete (i.e., P does not cover some positive example p)
      then generate a previously untried clause that covers p and add that clause to P;
    if P is not consistent (i.e., P covers some negative example n)
      then discard a clause from P that covers n
  until P is complete and consistent w.r.t. all examples read so far
forever
```

Induction is interactive (during the search for a false clause when *P* is found to be non consistent), via classification queries to the source. The generation of a new clause (in case of detected non-completeness) proceeds by general-to-specific refinement through the θ-subsumption-ordered lattice of clauses constrained by the biases. This results in intelligent pruning of the search space: if *P* is not complete w.r.t. some positive example, then no program more specific than *P* need be considered; conversely, if *P* is not consistent w.r.t. some negative example, then no program more general than *P* need be considered.

Example 15: During the synthesis of a program for the member predicate, the (here heavily annotated) dialogue between the technique and the source goes basically as follows:

MIS:	Declare the predicate for which a program is to be synthesised.
Source:	member(+x,+[x]). (The first argument is a term, while the second argument is a list of terms. The "+" means that both arguments must be ground at call-time.)
MIS:	What are the predicates possibly used by a member program? (That is: what is the basis set?)
Source:	member/2. (The source thinks there is a recursive program, using no other predicates.)
MIS:	The initial program P_0 is the empty program ■. First example?
Source:	member(a,[a]).
MIS:	P_0 is not complete, because member(a,[a]) is not covered. After a call to the clause generator, the revised program P_1 is {member(X,[Y|T]) ← }, which is consistent and complete w.r.t. the examples presented so far. Next example?
Source:	¬member(a,[b,c]).

4. We omit here the control aspects related to the detection of potential non-termination.

MIS: P_1 is not consistent, because member(a,[b,c]) is covered. The first, and only, clause of P_1 is found to be the false one and is thus removed. The resulting empty program is not complete, because member(a,[a]) is not covered. The clause generator needs more information and queries the source: Is member(a,[]) a positive example?

Source: No.

MIS: Now clause refinement goes through, and the revised program P_2 is {member(X,[Y]) ← }, which is consistent and complete w.r.t. the examples presented so far. Next example?

Source: ¬member(a,[b]).

MIS: P_2 is not consistent, because member(a,[b]) is covered. Its first, and only, clause is thus removed. The resulting empty program is not complete, because member(a,[a]) is not covered. After another call to the clause generator, the revised program P_3 is {member(X,[X|T]) ← }, which is consistent and complete w.r.t. the examples presented so far. Next example?

Source: member(c,[d,c]).

MIS: P_3 is not complete, because member(c,[d,c]) is not covered. The clause generator again needs to query the source: Is member(d,[c]) a positive example?

Source: No.

MIS: Clause refinement succeeds now, and the revised program P_4 is {member(X,[X|T]) ← , member(X,[Y|T]) ← member(X,T)}, which is consistent and complete w.r.t. the examples presented so far. Next example?

Suppose the source now stops presenting examples. The technique has synthesised a correct logic program. ♦

Like all incremental techniques, the technique is sensitive to the evidence ordering, and can be forced into the induction of infinite, redundant, or dead code. Also, it cannot perform any kind of predicate invention.

Many improvements of MIS have been proposed [26] [38][39] [52], and many variations thereof have been designed. Here we just list those that have been demonstrated essentially through their ability to synthesise recursive programs. The CONSTRUCTIVE INTERPRETER [23] is a passive variant, as it fully mechanises the oracle by requiring that a complete specification be adjoined to the example set. The MARKUS technique [12] [34] essentially improves on the clause generator. Some researchers have recognised that, as far as recursive programs are concerned, the search space could be considerably reduced if programs were constrained to fit certain schemas. The XOANON technique [77] is a variant of MIS that explores a second-order search space (a lattice, actually), ordered by a corresponding extension of θ-subsumption, with second-order expressions (called schemas) at the top, and first-order expressions (i.e., programs) at the bottom. Synthesis starts from a schema believed-to-be-applicable, and the improvement in synthesis time can be exponential when a "good" schema is selected. Similarly, the MISST technique [74] proposes a new clause generation operator for MIS, such that the inferred program corresponds to a *s*keleton to which programming *t*echniques (such as "adding a parameter") have been applied.

CIGOL, GOLEM, and PROGOL. The next three techniques are not really conceptually related, but were developed by overlapping teams.

The CIGOL ("logiC" read backwards) technique [57] is theory-guided, with evidence, initial theory, background knowledge, and hypotheses all as definite programs (i.e., no negative examples can be given). The evidence can be for possibly multiple relations, which can perform any kind of manipulations, but it has to be carefully chosen. No language or search bias can be given. The technique performs incremental, bottom-up theory-revision in order to complete the incomplete initial theory. It uses three inductive inference rules, discussed next. First, there is a restricted, non-deterministic form of the first-order absorption rule. The technique performs a best-first search for absorptions with a preference for "simple" consequent clauses. It uses Occam information compression to guide absorptions, using the total number of predicate and function symbol occurrences in the program as encoding measure. An absorption is then only allowed if it produces a positive compression value. Second, the technique can perform predicate invention, through a restricted, non-deterministic form of the first-order intra-construction rule. Notice that intra-construction itself can only induce *non*-recursive clauses for the invented predicate *q*. However, subsequent absorption on such clauses *can* introduce recursion into some clause for the invented predicate, hence making the technique able not only to perform pragmatic predicate invention but also necessary predicate invention. Third, the truncation rule is used for dealing with a boundary case of the intra-construction rule. It generalises a set of unit clauses by computing their lgθ. The technique is interactive in two ways. On the one hand, an oracle (which is the source here) is asked to recognise and name the invented predicates, given their abduced evidence.

On the other hand, generalised clauses obtained through absorption and truncation are tested by asking the oracle whether they are correct and even worthy to keep. The oracle thus has to be quite expert, both in computation and in the application domain. There is no restriction on the numbers of base clauses, recursive clauses, or recursive calls within the latter. The technique can however not detect or handle the sparseness problem, should it occur.

Example 16: Suppose we start from the empty initial theory and background knowledge. After being provided with the two clauses member(1,[1]) $\leftarrow$ and member(2,[2,3,4]) $\leftarrow$, a truncation is performed, yielding the clause member(X,[X|T]) $\leftarrow$, called c_1, which is validated as consistent by the source. Upon the next clause, say member(5,[4,5]) $\leftarrow$, another truncation takes place, giving member(X,[Y|T]) $\leftarrow$, which is however considered inconsistent by the source and thus not retained. After receiving another clause, say member(6,[5,6,7,8,9]) $\leftarrow$, a new truncation takes place, resulting in a clause considered consistent by the source, namely member(X,[Y,X|T]) $\leftarrow$. A subsequent absorption yields the recursive clause member(X,[Y|T]) $\leftarrow$ member(X,T), called c_2, which is also validated as consistent by the source. The source can now stop providing evidence, as clauses c_1 and c_2 constitute a correct implementation of the intended relation for member. ♦

In general, the technique seems to require less evidence than MIS, but it is also apparently at least as powerful as MIS (due to its additional predicate invention ability). Its incremental nature makes the technique quite sensitive to the evidence ordering.

The GOLEM technique [58] aims at overcoming the search explosion of its predecessor (due to the high non-determinism of the inverse resolution rules). The rlgθ operator is suitable for this purpose as it eliminates all search, though it suffers from its often unrealistic requirement for finite, ground background knowledge, from its induction of intractably large— if not infinite—clauses, and from its inability to induce multiple clause hypotheses (which is essential for recursive theories). All these problems are successfully tackled here, as described next.

The technique takes as evidence a set of ground literals (positive and negative examples) of one single relation that may perform any kind of manipulation, and as background knowledge a set of definite clauses where every head variable also occurs in the body. From such background knowledge, it is possible to generate a depth-limited Herbrand model, the integer depth bound h being provided by the source as a language bias. This model replaces the background knowledge when computing the rlgθ of two clauses, which makes the computation of the rlgθ possible (due to the groundness of the model), and the (reduced) rlgθ necessarily finite (due to the finiteness of the model). It now remains to make the (reduced) rlgθ tractably large, as its size normally grows exponentially in the number n of input clauses. It turns out that if the hypothesis language is restricted to ij-determinate definite clauses, then the number of literals in an rlgθ is upper-bounded by a polynomial function independent of n (the power being j^i). This bound is conservative, as practical observation shows no dramatic increase, and even an eventual decrease as n increases sufficiently. So it suffices that i and j are also provided as a language bias ($i = j = 2$ often is a good setting). Next, the original notion of clause reduction (through θ-subsumption equivalence) can be made even more effective through the optional provision of mode information as search bias and a clever usage of the negative examples. Finally, a multiple clause hypothesis can be induced even in an rlgθ-based technique, namely by iterating over an inner loop that computes a consistent rlgθ of only a subset of the given positive examples and retracting all covered positive examples at the end of each such iteration (hence a greedy control strategy).

The technique is passive, data-driven, non-incremental, and cannot perform any kind of predicate invention. The mentioned inner loop proceeds in three phases. In the first phase, a starting clause is chosen among the rlgθs of a random sampling of positive example pairs to be the consistent one covering the largest number of positive examples. In the second phase, the starting clause is greedily generalised in another loop, at each iteration taking its rlgθ with a positive example, chosen among a random sampling to be the one that yields the consistent rlgθ covering the largest number of positive examples, and this as long as the cover increases. Finally, in the third phase, the resulting clause is reduced. Recursion may appear in the rlgθ of clauses, depending on the background knowledge. There is thus no restriction on the numbers of base clauses, recursive clauses, or recursive calls within the latter.

The technique can induce the quicksort program from about 15 positive examples, 4 negative examples, and 84 literals in the generated model of the background knowledge, but it is generally awkward for inducing recursive programs [56]. It may even fail to induce a program that is correct w.r.t. the given evidence, even if there is one in its hypothesis space. It has been quite successful on real-world applications in knowledge acquisition and discovery, such as drug design and satellite fault diagnosis. It generally copes well with (very) large training sets.

The PROGOL (“Prolog” where the last three characters are inverted) technique [56] adds inductive inference rules to a standard Prolog interpreter, and can be run both in interactive mode and in passive mode. Its evidence language is Horn clauses, with a distinction between positive evidence (namely definite clauses) and negative evidence (negative examples plus integrity constraints, as their generalisation, i.e., headless Horn clauses). The evidence can be for possibly multiple relations, which can perform any kind of manipulations, but it has to be carefully chosen. The background knowledge is made of normal programs, including the standard Prolog primitives. Type, mode, and multiplicity information must be provided as search bias. By this token, one also declares which predicates can appear in the heads and which ones in the bodies of hypothesis clauses. Recursive clauses will be considered if some predicate may appear both in heads and in bodies. Other biases require the source to limit the number of clauses in a hypothesis, the depth *i* of variables and number *c* of literals in candidate clauses, the depth *h* of performed SLDNF resolutions, the number of clauses explored in the inner loop, etc., all this in order to ensure polynomial tractability of the technique. The hypothesis language is definite programs. There is no restriction on the numbers of base clauses, recursive clauses, or recursive calls within the latter.

The bottom-up technique applies a covering approach, which works as follows, starting from the empty hypothesis. After selecting the next piece of positive evidence, a consistent, bias-compliant clause covering it is generated (see below) and added to the hypothesis, after deleting from the latter all clauses made redundant by this new clause. Furthermore, *all* pieces of positive evidence that are covered by the new clause (i.e., not just the selected one) are deleted from the evidence. This is repeated until there are no more pieces of positive evidence. Inside this loop, the covering clause is constructed in a general-to-specific search in the θ-subsumption clause lattice, bounded on top by the empty clause and at the bottom by a clause $\bot_i$ (where *i* is the depth bound on the variables). The latter is the most specific finite and bias-compliant clause that can be computed from the chosen piece *e* of positive evidence and the background knowledge *B*, namely such that *e* is SLDNF-derivable from $B \wedge \bot_i$ in at most *h* steps. An A^*-like algorithm is used for the search of the covering clause, guaranteeing to return the clause that maximally Occam-compresses the current positive evidence, using the total number of atom occurrences in the program as encoding measure. (This does not mean that the final hypothesis maximally compresses the whole positive evidence, because the covered pieces of positive evidence are retracted at each iteration.) The technique cannot perform any kind of predicate invention. The covering approach with greedy search makes the technique very sensitive to the ordering of the evidence.

Example 17: Consider again the delOdds predicate of Section 3.1. Suppose given the following type, mode, and determinism declarations:

```
modeh(1,delOdds(+clist,–clist))       modeb(1,delOdds(+clist,–clist))
modeb(1, odd(+constant))              modeb(1,+clist=[–constant|–clist])
modeb(1,even(+constant))              modeb(1,–clist=[+constant|+clist])
```

These express that atoms of predicate delOdds may appear both in the heads (h) and in the bodies (b) of hypothesis clauses, with two variables of type constant-list (clist) as formal parameters, such that there is one (1) correct instance of such an atom if its first actual parameter is ground (+) and its second an unbound variable (–). Atoms of predicates odd and even may appear only in the bodies of clauses (hence no hypothesis is to be induced for them), with a variable of type constant as formal parameter, which must be ground at call-time. Equality (=) may also only appear in bodies, in atoms of the form L=[H|T], such that there is one (1) correct instance if either L or both H and T are ground at call-time, with the other variable(s) being unbound then. Let the other biases be set to their defaults ($c = 4$, $h = 30$, $i = 3$). Let the background knowledge contain the definitions of the involved types. Consider now the following evidence (ordered by column-wise traversal):

```
delOdds([],[]) ←                      delOdds([A],[]) ← odd(A)
delOdds([3,4],[4]) ←                  delOdds([B],[B]) ← even(B)
delOdds([3,5],[]) ←                   … examples of odd/1 …
delOdds([6,7],[6]) ←                  … examples of even/1 …
delOdds([6,8],[6,8]) ←
delOdds([7,8,9],[8]) ←                ← delOdds([1],[1])
delOdds([6,7,8,9],[6,8]) ←            ← delOdds([6,8],[8])
```

Note that 5 of the pieces of positive evidence for delOdds are identical to those used above by SYNAPSE, but that the other 2 pieces used by the latter are technically not necessary here, because they are subsumed by the 2 properties. The examples for odd and even are also new, as are the negative examples. From this evidence set, the technique automatically synthesises, in 3 iterations, the following correct program (which is similar to the one of Example 10, except that equality has here been eliminated through unfolding with the clause X=X ←):

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

Note that the evidence set had been carefully crafted (in order to be so small): dropping any piece of evidence will result in a wrong program. The properties replace large numbers of positive examples. ♦

FILP. The evidence language of the **FILP** (*Functional Inductive Logic Programming*) technique [7] [5] [6] is ground atoms (positive examples) for possibly multiple (syntactic or semantic manipulation) total functions, and they can be arbitrarily chosen. The intended relations must thus be total functions in some given modes, which must also be provided, as search bias. The hypothesis language is definite programs, where every predicate is used in a fully deterministic mode. The background knowledge is definite clauses, plus declarations of the modes of their head atoms. If some predicates in the background knowledge are only defined through ground unit clauses, then the latter are generalised as well. A language bias describing the hypothesis space is also given. An instance of such a bias for the sort predicate is:

```
sort(L,S) ← {L=[], S=[]}
sort(L,S) ← {L=[H|T], sort(T,V), insert(H,{V,S})}
```

The curly braces used for the body atoms and the parameters denote any subset of the elements inside them. For this bias, two sample described clauses are (taking one for each clause of the bias):

```
sort(L,S) ← L=[], S=[]
sort(L,S) ← L=[H|T], sort(T,V), insert(H,V,S)
```

Note that such a bias may indicate potential recursive calls. There is no restriction on the numbers of base clauses, recursive clauses, or recursive calls within the latter. We do not consider such a bias a schema, because it is problem-specific.

The technique consists of a clause generation loop that is repeated until all of the positive examples and none of the negative examples are covered by the generated clauses. Initially, every clause is a unit clause for a top-level predicate, where the parameters are all different variables. These clauses are clearly over-general. At each iteration, a literal is introduced to the body of the clause being specialised, by choosing among the possible literals of the language bias, in order to make the over-general clause cover fewer negative examples. The technique can thus not perform any kind of predicate invention. This addition of literals continues until the clause obtained does not cover any of the negative examples. During this addition of literals, if the clause does not cover any positive example, then backtracking occurs. Throughout the clause generation process, mode declarations are taken into account to reduce the search space, and negative examples are computed directly from the positive examples (by the closed world assumption), since the program being induced is supposed to be fully deterministic in the indicated mode (it thus suffices to keep the input values unchanged and to arbitrarily modify some output value). During the clause generation process, if there are missing positive examples, then they are asked from the oracle (which is the source here). In other words, the technique is interactive. For instance, let the generated clause be p(A,B) ← q(A,C), r(A,C,B), let the positive example being investigated to see if it is covered by that clause be +p(a,b), and let the background knowledge cover the atom q(a,c), but nothing for predicate r. Then, the oracle is queried for instantiation of the example r(a,c,X), and let the answer be r(a,c,b), hence allowing the proof that the positive example is covered by that clause.

Example 18: Suppose the examples +reverse([ ],[]), +reverse([a],[a]), +reverse([a,b],[b,a]), +reverse([a,b,c],[c,b,a]) are given. The background knowledge is a program of the append predicate. The language bias is reverse(X,Y) ← {X=[], Y=[], X=[H|T], …, reverse(T,W), …, append(W,[H],Y), …} (*sic*). Finally, the mode declarations append(in,in,out) and reverse(in,out) are given, with an implicit full determinism on these modes. The initial clause to be specialised is reverse(X,Y) ← . The first literal being added to the body of the clause is computed heuristically as Y=[]. However, the resulting clause covers the generated example –reverse([a],[]), so more literals need to be added. If the literal X=[H|T] is now added, then no positive examples are covered, so another literal has to be added instead. It is found to be X=[ ]. Now, the resulting clause reverse(X,Y) ← Y=[], X=[] covers just the example +reverse([],[]), which is removed from the example set. The second clause of the program is found in the same way, and is reverse(X,Y) ← X=[H|T], reverse(T,W), append(W,[H],Y). The two clauses above cover all positive examples, but no (inferred) negative ones. ♦

3.4 Summary

We now summarise our overview by means of a chart (see Table 1). The top five lines name classification criteria, whereas the bottom sixteen lines name actual comparison criteria and features, so that the techniques may be measured up to each other. In a cell, a cross (×) means that the feature is supported, no answer means that the feature is not supported, and “n/a” means that the question whether or not the feature is supported is non-applicable. The space allocated in this overview to a technique is (usually) proportional to its power in terms of the answers given to the comparison criteria and features.

Table 1: Comparison of techniques for the induction of recursive logic programs

	special-purpose recursion-only synthesisers												general-purpose				
	schema-biased								schema-less								
generality model [a]	ii	ii	θ	θ	θ	θ	θ	θ	ir	ir	θ	n/a	θ	θ	rθ	ie	n/a
interactive / passive [b]	p	i	p	p	p	p	p	i	p	p	p	p	i	i	p	i/p	i
data-driven / theory-guided [c]	dd	dd	dd	dd	dd	dd	dd	dd	tg	tg	dd	dd	tg	tg	dd	dd	dd
	CRUSTACEAN	CILP	FORCE2	SIERES	TIM	SYNAPSE	METAINDUCE	DIALOGS	SPECTRE II	MERLIN	SMART	SKILIT	MIS	CIGOL	GOLEM	PROGOL	FILP
evidence language [d]	gl	gl	gl	gl	ga	np	ga	np	gl	gl	gl	gl	gl	dp	gl	Hc	ga
# predicates in evidence	1	1	1	1	1	1	1	1	≥1	1	1	1	≥1	≥1	1	≥1	≥1
semantic manipulation?			×	×	×	×		×	×	×	×	×	×	×	×	×	×
from arbitrary evidence?	×	×	×	×	×			n/a			×	×			×		×
type information?								×			×	×	×			×	
mode information?				×	×	n/a		n/a			×	×	×		×	×	×
determinism information?						n/a		n/a					×			×	×
other bias? (except schemas)			×								×	×	×		×	×	×
background knowl. language [d]			gl	dp	dp	np		np			dp	dp	dp	dp	dp-	np	dp-
hypothesis language [d]	dp	dp	dp	dp	dp	np	dp	np	dp	dp	dp	dp	dp	dp	dp-	dp	dp-
# base clauses / predicate	1	1	1	?	1	≥1	1	≥1	≥1	≥1	1	≥1	≥1	≥1	≥1	≥1	≥1
# recursive clauses / predicate	1	1	1	1	1	≥1	1	≥1	≥1	≥1	1	≥1	≥1	≥1	≥1	≥1	≥1
# recursive calls / clause	1	1	1	≥1	1	≥1	1	≥1	≥1	≥1	≥1	≥1	≥1	≥1	≥1	≥1	≥1
necessary predicate invention?		×		×		×	×	×		×				×			
useful predicate invention?						×		×									
handling of sparseness?	×	×						×			×						

a. θ = θ-subsumption, rθ = relative θ-subsumption, ir = inverse resolution, ii = inverse implication, ie = inverse entailment
b. i = interactive, p = passive
c. dd = data-driven, tg = theory-guided
d. ga = ground atoms, gl = ground literals, dp = definite programs, dp- = some restricted form of dp, np = normal programs, Hc = Horn clauses

An important remark is that all overviewed special-purpose techniques are non-incremental, so that the distinction of bottom-up versus top-down induction does not really apply to them, nor the consideration of identification criteria. There is no theoretical reason preventing the design of incremental special-purpose techniques for inducing recursive hypotheses, but it is nevertheless noteworthy that all known (to us) techniques of inductive synthesis of recursive programs are non-incremental. Incrementality thus does not seem to be a promising research avenue for this field. Indeed, as seen, incremental techniques are often very sensitive to the ordering of the evidence, in the sense that infinite, redundant, or dead code may be generated (from an adverse ordering). Such behaviour is probably deemed dangerous, as incremental techniques do not really have a sense of direction when they are synthesising recursive programs, which is not really adequate considering the fragile nature of recursive programs.

Also, the table clearly identifies (other) directions for future research, as well as cross-fertilisation opportunities, such as the combination of the best ways of satisfying a criterion in order to design new techniques. No technique is intrinsically superior to all others, and further judgment depends on the application setting (see the next section).

4 Prospects of Inductive Synthesis

In the previous section, we have discussed the achievements of the inductive synthesis of recursive programs in an application-independent fashion. We now discuss the application prospects of such techniques, be they the existing ones or forthcoming ones. From our doubts on the realism of some existing techniques for the intended application area, we filter out more directions for future research and assess the viability of inductive synthesis in that area.

There are essentially two such application areas. The first, *knowledge acquisition and discovery* (Section 4.1), has actually never been explicitly targeted by research on inductive synthesis of recursive programs, but we have some thoughts here. The second, *software engineering* (Section 4.2), is the most frequently targeted one, but has been the object of much controversy and prejudice, which we summarise and then support or debunk, as necessary.

4.1 Applications in Knowledge Acquisition and Discovery

Knowledge acquisition and discovery from data (and data mining) is about extracting and transforming hidden information into valuable knowledge through the discovery of relationships and patterns in these data. This sounds very much like a vague re-formulation of the general ILP task itself, but we here consider it an application area as the data in question is usually very voluminous. In fact, this is a very natural application area for ILP and we expect ILP to have its most impressive results here, especially that such has already been the case anyway. So there is no need to argue as far as ILP as a whole is concerned.

But what about the usefulness of inductive synthesis of recursive programs to this application area? Especially that, intuitively, just like the procedures in application software, very few real life concepts seem to have recursive definitions, rare examples being ancestor and natural language. We argue that it is worth having a special-purpose recursion synthesiser *attached to* a general-purpose induction technique. Indeed, a general-purpose technique may detect (or conjecture) the necessity (or usefulness) of inventing a new predicate, and since such a new predicate is then known in advance to have a recursive program (see Section 2.4.2), it seems preferable to invoke a special-purpose recursion synthesiser for such auxiliary purposes rather than have the general-purpose technique do it all. Also, the (abduced) evidence of the new predicate may be quite sparsely distributed over its intended relation, and (some) special-purpose recursion synthesisers handle this situation quite well (as seen in Section 3), whereas even the best general-purpose techniques require a lot of evidence in order to reliably induce recursively defined hypotheses: see the experiments with MIS in [68] [69], with FOIL in [65] [66], or with PROGOL in [56]. We believe that general-purpose ILP techniques and special-purpose recursion synthesisers have both much to gain from such a synergy.

4.2 Applications in Software Engineering

Wouldn't it be nice if we could automatically obtain correct programs from specifications consisting just of a few examples of their input/output behaviour, or would it? This dream of automa-g-ic programming is as old as Computer Science and has been an area of intense research since the late 1960s. As there is no difference between (executable) formal specifications and programs [51], this is sometimes called *programming by examples* and can be seen as an innovative programming technique, especially aimed at two categories of programmers:

- *expert programmers* would often rather just provide a few carefully chosen examples and have a synthesiser work out the details (of recursion) for them, hence increasing their productivity;
- *end users* are often computationally naive and cannot provide (much) more than examples, but this should nevertheless allow them to do some basic programming tasks [21], such as the recording of macro definitions, etc.

Of course, any programmer in the spectrum laid out by these extremes can benefit from programming by examples, but we believe that the risk/benefit ratio is optimal for these extremes of expertise. Indeed, the risk is that an incorrect program can be synthesised. This risk can be minimised by an expert user who knows how the synthesiser works and how reliable it is. The risk is not so relevant for end users, as they usually do not want to write safety-critical software anyway and can thus cope with approximate programs.

In any case, the scenario here is that the source of all inputs is a human (called the *specifier*, though we may also speak of the *programmer*), and this has to be taken into account as well as exploited. Indeed, a human cannot be expected to provide inputs (called the *specification*) that are too voluminous, especially that an expert programmer would thus actually lose in productivity. Also, a human has considerably more expertise than the average source and oracle, and this may be exploited, say in an interactive fashion. The specifier also is the oracle (if any).

The scenario also requires an extremely high (ideally 100%) accuracy of the synthesised program against the test set if not against the entire intended relation, because a program that does not exactly do what is expected is useless (though this may not be a big problem in end user computing). The slightest mistake in a recursive clause is usually amplified manifold through recursion before a base clause becomes applicable.

Since one does not in general know in advance whether a recursive program exists or not, we suggest (in case of doubt) to first invoke a recursion synthesiser and fall back onto a general technique if the former fails. This is a suitable invocation scenario for programming assistance applications, as for many problems a suitable recursive (e.g., divide-and-conquer) program will be much more efficient than any non-recursive (e.g., generate-and-test) program. Actually, even during induction by a general-purpose technique, necessary (or useful) invention of a new predicate may be detected or conjectured: the general-purpose technique could then invoke a recursion synthesiser, since the new predicate is then known in advance, by definition, to have a recursive program (see Section 2.4.2).

We will here only discuss the prospects of techniques for the inductive synthesis of recursive programs for program construction, but not for related tasks, such as program verification [4] [6] [12], program transformation [14], etc. Any ILP technique may of course be interfaced with a program transformer (which reduces the time/space complexity and/or increases the time/space efficiency of programs), since a program to be transformed may have been synthesised by any approach, be it deductive, constructive, inductive, manual, mixed, sorcery, or whatever.

We urge the reader to remember at all moments that the discussion below is *only* about the programming assistance application area, but not about all mentioned application areas within software engineering: any criticism should be application-specific. See [7] for another discussion of applying ILP techniques to software engineering.

4.2.1 The Background Knowledge Usage Bottleneck

Some researchers have been wondering about interfacing ILP with deductive/constructive synthesis, so that these tasks be complementary rather than competing. Indeed, since the latter assumes given a formal specification, the question arises where such a specification would come from. Such knowledge acquisition tasks have been successfully tackled by ILP techniques for building the knowledge base of expert systems, but can ILP help here as well? Since specifications are usually required to be non-recursive (representing often a naive and inefficient program, for instance of the generate-and-test class), the techniques overviewed here do not apply and inducing such specifications would be a general ILP task. However, we believe that it is even more time-consuming and risky (but not more difficult) to induce generate-and-test programs from incomplete information than to synthesise recursive (e.g., divide-and-conquer) programs from such information. Indeed, the class of generate-and-test programs has so little structure, as opposed to the class of divide-and-conquer programs (remember the schema of Example 1), that we see no way how the induction of generate-and-test programs could be efficiently and effectively guided. Just consider the potentially huge set of background knowledge predicates.

This brings us directly to a first problem of many existing inductive synthesisers, namely their *background knowledge usage bottleneck* [32]. In a realistic programming scenario, the background knowledge consists of clauses for numerous predicates, just like with human programmers. However, we humans tend to dynamically organise this background knowledge according to relevance criteria, so that we do not think of using a definition of the grand-mother concept when constructing a sorting program. Or, less dramatically, during the construction of a quicksort program for integer lists, background knowledge about binary tree processing or lexicographic ordering of characters tends to be more in the background than knowledge about list processing, and, at one point during that construction, even knowledge about list merging or splitting may move further back.

Many researchers have tried to simulate this human hierarchy of background knowledge, though often in a very extreme way: transcripts (e.g., FORCE2 [18, p.78], TIM [40], [59, p.633], etc.) are shown where the background knowledge contains *exactly and only* some predicates actually sufficient (up to necessary predicate invention) to complete a synthesis. For instance, when the evidence is about sort, then partition and append are put into the background knowledge, and a quicksort program is induced. This is certainly a fine result, but there are two problems with it.

First, it only establishes the inducability of such a program by these techniques in an *optimal* scenario. But what about the monotonicity of inducability: if we add merge and split to that background knowledge, will the techniques still be able to induce the quicksort program? Will they find a merge-sort program? Will they find other sorting programs? What about the efficiency of induction: will they find all these programs quickly? What if we add potentially irrelevant predicates, such as arithmetic operations: are monotonicity and efficiency of induction preserved? Will the techniques discover (efficient) new sorting programs? Is useful predicate invention performed to avoid undisciplined background knowledge usage? Does the ordering of the background knowledge affect the synthesised program? The problem thus is that the optimal scenario is often unrealistic: in general, one does not know in advance which parts of the background knowledge will be relevant during a synthesis. One can make educated guesses, but creativity has its own ways. Finally, if one has to manually select the potentially relevant background knowledge before every synthesis session, then a poor productivity (at least of expert users) will be achieved. We thus believe in the following recommendation: *Within a given problem domain, background knowledge should be problem-independent and given once and for all (rather than crafted for each session), and the induction technique should dynamically order it.*

Second, such a scenario amounts to actually *teaching* a quicksort program, which goes counter specification practice: one specifies *all* possible programs for a problem (and how to use them), but not *a* possible program. Now we come to the earlier (in Section 2.1) announced justification of why the teacher and learner terminology is sometimes misleading and why we decided to speak of source and induction technique instead: a teacher (usually) knows how the taught concept can be defined, whereas a specifier does not always know how the specified problem can be implemented (recursively). Choosing between the teacher/learner and the specifier/synthesiser terminologies is thus application-specific, and neither terminology applies to induction as a whole. One may of course argue for the higher realism of the scenario where only *potentially* (rather than actually) relevant predicates are placed into the background knowledge, because the source then really is a specifier rather than a teacher. However, as argued above, this approach only works if such potentially relevant predicates are stored in a problem-independent domain-specific collection that can be designated by name rather than enumerated by hand for each synthesis.

In any case, this discussion shows that much research is needed in order to more effectively simulate the human ability of dynamically organising problem-specific background knowledge according to its relevance to the particular problem at hand, and even to the stage of solving that problem. This is called *knowledge mobilisation* by Polya [64]. In a first approximation, there need not be much focus on simulating creativity (algorithm discovery). A promising direction seems to be the pre-determination of the dynamic relevance ordering for a class of programs, so as to partition background knowledge predicates according to their relevance to the place-holders of a template capturing that class, and according to the types of their parameters [28] [30] [42]. This approach even has the advantage of being also useful for the re-use problem in deductive/constructive synthesis. Another, complementary approach is to try to *avoid* background knowledge usage in certain well-defined situations (such as the induction of the compose predicate of a divide-and-conquer program), namely by useful predicate invention (see Example 7 in Section 2.4.2 for a description of the problems that occur when useful predicate invention is avoided). Maybe background knowledge usage should, in such situations, only be done when such predicate invention fails?

4.2.2 Other Occurrences of the Knowing-an-Answer Syndrome

There are other occurrences of the knowing-an-answer syndrome, which is incarnated when running a synthesiser in the teacher/learner setting rather than in the specifier/synthesiser setting. In general thus, the symptoms of this syndrome are that *a* possible program is somehow subtly encoded in the inputs (evidence, background knowledge, bias, or initial theory), hence making inductive synthesis a mere extraction process. We now discuss this syndrome when the encoding is done in inputs other than the background knowledge, though the problematic consequences are the same as discussed above.

Some techniques require the source to know the base clauses of a possible program, in the sense that they have to be somehow provided in the inputs (e.g., the *basecase* function of FORCE2 [18]), possibly because the technique can only induce the recursive clauses. However, note that the base clauses of all possible programs for a predicate are *not* the same: for the sort predicate, an insertion-sort program has one base-clause (for the empty list), whereas a merge-sort program (with splitting of the list into two halves) has two base clauses (one for the empty list, the other for singleton lists).

Other techniques even require the source to know the recursive clauses of a possible program, in the sense that the provided examples must be on the same resolution path in order for the technique to find such a recursive clause (e.g., LOPSTER [47]). This implies that the evidence cannot be arbitrarily chosen, but must be carefully crafted, having a possible solution strategy in mind. This restriction can sometimes be overcome by using inverse implication or inverse entailment generalisation models.

Still other techniques require the source to encode an *entire* possible program in a language bias. For instance, the clause description language of FILP [5] allows the following language bias (note that it is but a slight variant of the one in Section 3.3):

```
sort(L,S) ← {X=[], Y=[]}
sort(L,S) ← {X=[H|T], sort(Y,V), insert(E,W,R)}
```

but FILP cannot infer the correct dataflow by unifying some of its variables. In other words, a correct dataflow has to be given, as one cannot just list the potentially useful predicates. So let us give a correct dataflow and see what happens when not all the computations are given. The following bias is unfortunately in general insufficient (note its similarity now to a problem-independent divide-and-conquer template, see Example 1):

```
sort(L,S) ← {L=[], solve(S)}
sort(L,S) ← {L=[H|T], sort(T,V), compose(H,V,S)}
```

as FILP cannot induce programs for the problem-dependent solve and compose predicates, unless they are described in the evidence or background knowledge, which would however return the argument to the background knowledge usage bottleneck (in Section 4.2.1). In other words, correct computations have to be given as well. Overall thus, a FILP bias gives very little else beyond a correct program.

Similarly for the algorithm sketches of SKILIT [13]: although they do not necessarily give away an entire program, they often reveal much of a possible program. Fortunately, the technique also works from self-constructed blackbox sketches, which happens when it is given no user-provided sketches.

Finally, the basis set of MIS [69] and the mode declarations of PROGOL [56] amount to hand-picking, for each problem, the believed-to-be-relevant predicates (but not their arguments, as for FILP above) from a potentially large background knowledge, which may however not be suitable to all kinds of programmers. The inability of these techniques to override this bias or even to invent new predicates further hampers the programmer.

In all these techniques, the idea is that the specifier should somehow be computer-assisted when s/he has an approximate idea of a possible solution strategy. However, this reduces the productivity of the (expert) specifier and the creativity of the synthesiser, but may of course be interesting in some cases. Also note that, for non-recursively definable concepts, from a given viewpoint, there is usually only one correct description. For instance, for the bird concept, there is one description from a cat's point of view, one description from a biologist's point of view, etc. But not so for recursively definable concepts, where there are usually many (even viewpoint-independent) correct programs [32]. For instance, for the sort predicate, there are programs implementing the quicksort algorithm, the merge-sort algorithm, etc.

4.2.3 The Background Knowledge Usage Miracle

Some techniques feature another problem with background knowledge usage, namely that certain predicates *must* be selected from it in order to induce a program (unless they are invented), no matter what algorithm is implemented by that program. We call such predicates *intrinsic predicates* (to a problem), as opposed to *extrinsic predicates*, which need not appear in a program (for that problem). For instance, if the evidence of sort does not mention the ≤ predicate for deciding the total order according to which the elements have to be sorted, then that predicate *must* somehow be selected from the background knowledge (unless it is invented or intrinsic to some used background knowledge predicate), whether the final program is a quicksort or a merge-sort program. It is thus intrinsic to sort. However, the append predicate is extrinsic to sort, because it appears in a quicksort program but not in a merge-sort program. If intrinsic predicates are not invented, then we consider it a *miracle* if they are actually selected from the background knowledge. (Note that the bottleneck problem is thus about the presence of too many extrinsic predicates, whereas the miracle problem is about intrinsic predicates.) Such a miracle may (have to) happen in a general ILP setting, but is useless in a programming setting, where the specifier is a human being. Indeed, no human specifier can possibly want a program for sort without knowing the ≤ predicate: such an intrinsic predicate is not peculiar to the specifier's mental sorting algorithm (if s/he has any), but proper to the sorting *problem*.

So the specifier should somehow be able to convey the intrinsic predicates to the synthesiser, so as to avoid that the synthesiser has to spend time on predicate invention or on guesswork among the background knowledge. With specifications by examples only, conveying such intrinsic predicates is impossible. There are at least two related, complementary approaches to overcoming this problem. First, the evidence language can be extended, for instance to (non-recursive) Horn clauses [28] [43] [56], or even to general clauses [22]. Second, synthesis can be interactive, asking the specifier questions in whose answers the intrinsic predicates (if any) must appear [30]. Note that the provision of the intrinsic predicates does not mean a productivity loss for the specifier, because s/he ought to know these intrinsic predicates anyway and need not give the extrinsic ones, but thus rather a reliability and productivity gain for the synthesiser.

4.2.4 Scenario Violations: Too Voluminous/Sophisticated Inputs, Too Inaccurate Programs, etc.

Some techniques violate the scenario laid out above, in the sense that they require too *voluminous* inputs (such as the problem-specific biases and initial theories of FILP [5], SPECTRE II [9], MERLIN [10], PROGOL [56], and MIS [69]) from some categories of specifiers, or induce programs that have too low accuracies against training or test sets (e.g., SKILIT [42], GOLEM [58]), or even that have both problems (e.g., SKILIT+MONIC [43], which is surprising as one would conjecture that many inputs mean high accuracies). (We ignore here the already discussed problems when the background knowledge and biases are manually tuned for a given problem.) It is of course very subjective to define what is meant by too voluminous inputs and too inaccurate programs, especially that they are related issues. We estimate that a viable technique should synthesise an *n*-literal program from specifier-provided inputs of maximum $c \cdot n$ literals (or words), with a (nearly) 100% accuracy against an arbitrary test set, where *c* varies between 1 (for experts) and 5 (for end users). In this sense, most here overviewed techniques have too voluminous inputs, especially those requiring a manual encoding of (part of) a possible program in a bias. To us, it seems that, just like for background knowledge, *biases should be problem-independent, within a given problem domain* (note that such is already the case for schemas, by their very definition). Fortunately, most techniques overviewed here synthesise quite accurate programs in this sense, and this usually even from very little evidence.

A related violation is the requirement of too *sophisticated* inputs (such as the *basecase* and *maxdepth* functions of FORCE2 [18], the validation queries of CIGOL [57], or the problem-specific biases and initial theories of FILP [5], SPECTRE II [9], MERLIN [10], SKILIT [42], SKILIT+MONIC [43], PROGOL [56], and MIS [69]). An end user cannot always be able to provide adequate biases (and background knowledge), and an expert user would be slowed down by providing such inputs. Also, some theory-guided induction techniques put tight pre-conditions on the initial theory (e.g., SPECTRE II [9], MERLIN [10]), which may be hard to ensure even by expert users. For instance, SPECTRE II [9] imposes that there are no positive and negative examples that have the same sequence of input clauses in their refutations, which is an undecidable property. Or, the initial theory may be required to be overly general, rather than in an arbitrary connection to the intended relations. Over-generality is fortunately easy to establish (and is thus quite general [16]): it suffices to use a template as the initial theory. A template like the one in Example 1 is inadequate because its predicate symbols are meant to be open, but one can specialise it in a problem-specific fashion so that it is guaranteed to be overly general, for instance as follows (assuming the list predicate is known):

```
sort(L,S) ← L=[], list(S)
sort(L,S) ← L=[H|T], sort(T,V), list(V), list(S)
```

Unfortunately, many theory-guided techniques cannot cope with such an initial theory (which is almost a template).

4.2.5 Information Loss

Some techniques feature *information loss* during the induction process, and this is especially dramatic in a programming context (where high accuracy is crucial), though deplorable in any case.

For instance, for near-minimal-sized evidence sets, the behaviour of PROGOL [56] may become quite unpredictable. When retrying a successful synthesis after *adding* a piece of evidence, the induction sometimes fails (e.g., insert the new, innocuous-looking, and even redundant clause delOdds([1],[]) ← into second position of the positive evidence in Example 17). Conversely, when retrying a failed synthesis after *dropping* a piece of evidence, the synthesis sometimes is successful. (Failure and success are here judged according to whether the inferred program is incorrect or correct.)

Also, we have already mentioned the sensitivity to evidence ordering of incremental techniques. PROGOL [56] can succeed or fail on the same training set, depending on the order in which it is presented (e.g., moving the properties of Example 17 forward by at least four positions results in an incorrect program). MIS [69] can even be forced into the induction of infinite, redundant, or dead code.

Finally, for the induction of a program for union(A,B,C), which holds iff set C is the union of sets A and B, the SKILIT technique [42] is reported in [43] to have an accuracy of 22.5%±6.1 from 10 randomly generated positive examples and 0 negative examples, but an accuracy of only 18.6%±5.3 from 10 positive and 10 negative examples. This accuracy loss from more information can be explained as follows: synthesis with no negative examples tends to produce more general programs than with negative examples, so that accuracy on a positive test set may be higher. Also, SKILIT+MONIC [43] results in rather low accuracies, even when starting from correct and complete information in the integrity constraints. For instance, from integrity constraints with correct and complete information as well as 20 randomly generated positive examples for the union predicate, the accuracy is only 47.6%±35.0. The technique fortunately has the advantage of still working from incomplete information in the integrity constraints (as it does not know how complete their information is), but then the resulting accuracies might drop even lower, e.g., to levels where negative examples are used.

In general, it seems that constructive ways of using negative evidence (when it is labelled as such) have not been properly explored: when induction is driven by the positive evidence (as for SKILIT), then the negative evidence (or the constraint set) is often only used for an analytico-destructive purpose, namely the acceptance or rejection of a candidate program. However, especially when negative evidence is given as (Horn-)clausal constraints [22] [23] [28] [43] [56], it should be possible to use it constructively as well. To the best of our knowledge, only SYNAPSE [28] and the CONSTRUCTIVE INTERPRETER [23] do so (and in quite similar ways).

4.2.6 Conclusion about Programming Applications

We repeat that we do not mean to imply that the techniques discussed here are useless in general, but only that they are sometimes unrealistic (at least in their current versions) for real programming assistance applications.

Progress has been very slow (even negligible according to some) in this application area (if one considers all tackled target languages), and, after nearly 30 years of research without much practical results, the legitimate question arises whether research should be continued at all in this field. Perhaps symptomatically, the European Union-sponsored project ILP-2 (the follow-up to the ILP project of ESPRIT III) does not cover software engineering applications. There has been significant controversy and prejudice [32] about the usefulness of such research, even and especially outside the community. Insider detractors may point to the problems raised in this section, and we of course support such warnings, whereas outsider detractors usually raise the risk issue, which we would however like to debunk [32]: when applicable, inductive synthesis is no more risky than deductive/constructive synthesis! Indeed, the only difference is that the former starts from known-to-be-incomplete information and the latter from assumed-to-be-complete information, but in *both* cases one has *no* guarantee that the synthesised program does what was actually *intended*. That deductive/constructive synthesis guarantees that the synthesised program does what was *specified* does not affect the fact that it is the formalisation step from intentions to formal specifications that is risky, rather than the *kind* of synthesis being performed from the produced specification. The main issues are that a specification should be labelled as probably-incomplete or potentially-complete, and that an appropriate kind of synthesis technique should be invoked. The two approaches can thus be considered complementary, rather than rivals, and the ultimate decision of which one to use should lie with the specifier, not with the research community.

So then, what is our statement on the future of the inductive synthesis of recursive programs applied towards programming assistance? We believe such techniques *can* be (made) viable, provided more research is done on overcoming the obstacles listed above, provided more realistic programming scenarios are aimed at, and provided the future work directions and cross-fertilisation opportunities of Table 1 are pursued. We believe that some categories of programmers *would* use such techniques, provided it improves their productivity or increases the class of programs they can write by themselves.

5 Conclusion

The inductive synthesis of recursive (logic) programs is a challenging and important sub-field of ILP. It is challenging because recursive programs are particularly delicate mathematical objects that must be designed with utmost care. It is important because recursive programs (for certain predicates) are sometimes the only way to complete the induction of a finite hypothesis (involving these predicates). We have overviewed the achievements of this sub-field, throwing in theoretical results and historical remarks where appropriate. These achievements, after over a quarter-century of research, are a clear testimony to the difficulty of the task: witness the slow progress in increasing synthesis reliability and speed, and in decreasing the volume and sophistication of the required inputs; also witness the huge variety of different approaches. We have also debated the practical applicability of the overviewed techniques in two application areas, namely knowledge discovery and software engineering (or rather: programming). It turns out that these are completely different settings and that such settings (may) have to be exploited and taken into account when designing new techniques. We are confident that there *is* a future for such techniques (especially that they are necessary anyway), provided progress is made by combining the best individual results into powerful and reliable inductive recursion synthesisers.

Acknowledgments

The first author thanks some of the ILP'96 and LOPSTR'96 participants for stimulating discussions on the feasibility, necessity, and future of the inductive synthesis of recursive programs, which conversations eventually led to the decision to write this paper. Both authors gratefully acknowledge the assistance of some of the designers of the techniques discussed here, for providing us with implementations and missing papers, and for patiently answering our probing questions. We are also indebted to the anonymous referees, to the editors of this special issue, as well as to Lubos Popelínsky and Alípio Jorge, for their insightful comments on the first versions of this paper.

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] D. Angluin and C.H. Smith. Inductive inference: Theory and methods. *Computing Surveys* 15(3):237–269, Sept. 1983.

[4] F. Bergadano *et al*. Inductive test case generation. In S. Muggleton (ed), *Proc. of ILP*'93, pp. 11–24. Technical Report IJS-DP-6707, J. Stefan Institute, Ljubljana, Slovenia, 1993.

[5] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In R. Bajcsy (ed), *Proc. of IJCAI*'93, pp. 1044–1049. Morgan Kaufmann, 1993.

[6] F. Bergadano and D. Gunetti. Inductive synthesis of logic programs and inductive logic programming. In Y. Deville (ed), *Proc. of LOPSTR*'93, pp. 45–56. Springer-Verlag, 1994.

[7] F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering*. The MIT Press, 1995.

[8] A.W. Biermann. Automatic programming. In S.C. Shapiro (ed), *Encyclopedia of Artificial Intelligence*, second, extended edition, pp. 59–83. John Wiley, 1992.

[9] H. Boström. Specialization of recursive predicates. In *Proc. of ECML*'95. LNAI, Springer-Verlag, 1995.

[10] H. Boström. Theory-guided induction of logic programs by inference of regular languages. In *Proc. of ICML*'96. Morgan Kaufmann, 1996.

[11] H. Boström and P. Idestam-Almquist. Specialization of logic programs by pruning SLD-trees. In S. Wrobel (ed), *Proc. of ILP*'94, pp. 31–48. GMD-Studien Nr. 237, Sankt Augustin, Germany, 1994.

[12] I. Bratko and M. Grobelnik. Inductive learning applied to program construction and verification. In S. Muggleton (ed), *Proc. of ILP*'93, pp. 279–292. TR IJS-DP-6707, J. Stefan Inst., Ljubljana, Slovenia, 1993.

[13] P.B. Brazdil and A.M. Jorge. Learning by refining algorithm sketches. In A. Cohn (ed), *Proc. of ECAI*'94, pp. 443–447. John Wiley, 1994.

[14] M. Bruynooghe and D. De Schreye. Some thoughts on the role of examples in program transformation and its relevance for explanation-based learning. In K.P. Jantke (ed), *Proc. of AII*'89, pp. 60–77. LNCS 397, Springer-Verlag, 1989.

[15] W. Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence* 36(2):149–176, Sept. 1988.

[16] W.W. Cohen. The generality of over-generality. In *Proc. of IWML*'91, pp. 490–494. Morgan Kaufmann, 1991.

[17] 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.

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

[19] W.W. Cohen. PAC-learning recursive logic programs: Efficient algorithms. *Journal of Artificial Intelligence Research* 2:501–539, 1995.

[20] W.W. Cohen. PAC-learning recursive logic programs: Negative results. *Journal of Artificial Intelligence Research* 2:541–573, 1995.

[21] A. Cypher. EAGER: Programming repetitive tasks by example. In *Human Factors in Computing Systems, Proc. of CHI'*91, pp. 33–39. ACM Press, 1991.

[22] L. De Raedt and M. Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence* 53(2–3):291–307, Feb. 1992.

[23] 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.

[24] Y. Deville. *Logic Programming*: *Systematic Program Development*. Addison Wesley, 1990.

[25] Y. Deville and K.-K. Lau. Logic program synthesis: A survey. *Journal of Logic Programming, Special Issue on 10 Years of Logic Programming,* 19–20:321–350, May/July 1994.

[26] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H. Rogers (eds), *Proc. of META'*88, pp. 501–521. The MIT Press, 1988.

[27] E. Erdem and P. Flener. Completing open logic programs by constructive induction. *International Journal of Intelligent Systems*, 1999.

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

[29] P. Flener. Predicate invention in inductive program synthesis. Technical Report, 1995. (Available as http://www.csd.uu.se/~pierref/pub/TRpredInv.ps.gz)

[30] P. Flener. Inductive logic program synthesis with DIALOGS. In S. Muggleton (ed), *Proc. of ILP'*96, pp. 175–198. LNAI 1314, Springer-Verlag, 1997.

[31] 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.

[32] 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.

[33] P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In: N.E. Fuchs (ed), *Proc. of LOPSTR'*97, pp. 124–143. LNCS 1463, Springer-Verlag, 1998.

[34] M. Grobelnik. Induction of Prolog programs with Markus. In Y. Deville (ed), *Proc. of LOPSTR'*93, pp. 57–63. Springer-Verlag, 1994.

[35] M. Hagiya. Programming by example and proving by example using higher-order unification. In M.E. Stickel (ed), *Proc. of CADE'*90, pp. 588–602. LNCS 449, Springer-Verlag, 1990.

[36] M. Hagiya. From programming-by-example to proving-by-example. In T. Ito and A.R. Meyer (eds), *Proc. of TACS'*91, pp. 387–419. LNCS 526, Springer-Verlag, 1991.

[37] 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.

[38] M.M. Huntbach. An improved version of Shapiro's Model Inference System. In E.Y. Shapiro (ed), *Proc. of ICLP'*86, pp. 180–187. LNCS 225, Springer-Verlag, 1986.

[39] M.M. Huntbach. Program synthesis by inductive inference. In B. du Boulay, D. Hogg, and L. Steels (eds), *Proc. of ECAI'*86, pp. 335–344. Elsevier, 1987.

[40] P. Idestam-Almquist. Efficient induction of recursive definitions by structural analysis of saturations. In L. De Raedt (ed), *Advances in Inductive Logic Programming,* pp. 192–205. IOS Press, 1996.

[41] A.M. Jorge and P.B. Brazdil. Exploiting algorithm sketches in ILP. In S. Muggleton (ed), *Proc. of ILP'*93, pp. 193–203. Technical Report IJS-DP-6707, J. Stefan Institute, Ljubljana, Slovenia, 1993.

[42] A.M. Jorge and P.B. Brazdil. Architecture for iterative learning of recursive definitions. In L. De Raedt (ed), *Advances in Inductive Logic Programming*, pp. 206-218. IOS Press, 1996.

[43] A.M. Jorge and P.B. Brazdil. Integrity constraints in ILP using a Monte Carlo approach. In S. Muggleton (ed), *Proc. of ILP'*96, pp. 229–244. LNAI 1314, Springer-Verlag, 1997.

[44] J.-P. Jouannaud and Y. Kodratoff. Characterization of a class of functions synthesized from examples by a Summers-like method using the Boyer-Moore-Wegbreit matching technique. In *Proc. of IJCAI'*79, pp. 440–447.

[45] T. Kanamori and H. Seki. Verification of Prolog programs using an extension of execution. In E.Y. Shapiro (ed), Proc. of ICLP'86, pp. 475–489. LNCS 225, Springer-Verlag, 1986.

[46] Y. Kodratoff and J.-P. Jouannaud. Synthesizing LISP programs working on the list level of embedding. In A.W. Biermann, G. Guiho, and Y. Kodratoff (eds), *Automatic Program Construction Techniques*, pp. 325–374. Macmillan, 1984.

[47] S. Lapointe and S. Matwin. Sub-unification: A tool for efficient induction of recursive programs. In *Proc. of ICML*'92, pp. 273–281. Morgan Kaufmann, 1992.

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

[49] N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.

[50] G. Le Blanc. BMW_k revisited: Generalization and formalization of an algorithm for detecting recursive relations in term sequences. In F. Bergadano and L. De Raedt (eds), *Proc. of ECML*'94, pp. 183–197. LNAI 784, Springer-Verlag, 1994.

[51] B. Le Charlier and P. Flener. Specifications are necessarily informal, or: Some more myths of formal methods. *Journal of Systems and Software* 40(3):275–296, March 1998.

[52] Y. Lichtenstein and E.Y. Shapiro. Abstract algorithmic debugging. In R.A. Kowalski and K.A. Bowen (eds), *Proc. of ICLP'*88, pp. 512–531. The MIT Press, 1988.

[53] J. Marcinkowski and L. Pacholski. Undecidability of the Horn-clause implication problem. In *Proc. of the 33rd IEEE Annual Symposium on Foundations of Computer Science*, pp. 354–362. 1992.

[54] C.R. Mofizur and M. Numao. Top-down induction of recursive programs from small number of sparse examples. In L. De Raedt (ed), *Advances in Inductive Logic Programming*, pp. 236–253. IOS Press, 1996.

[55] S. Muggleton. Inductive logic programming. *New Generation Computing* 8(4):295–317, 1991. Also in S. Muggleton (ed), *Inductive Logic Programming*, pp. 3–27. Volume APIC-38, Academic Press, 1992.

[56] S. Muggleton. Inverse entailment and Progol. *New Generation Computing* 13:245–286, 1995.

[57] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proc. of ICML'*88, pp. 339–352. Also in S. Muggleton (ed), *Inductive Logic Programming*, pp. 261–280. Volume APIC-38, Academic Press, 1992.

[58] S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton (ed), *Inductive Logic Programming*, pp. 281–298. Volume APIC-38, Academic Press, 1992.

[59] 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.

[60] C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in inductive logic programming. In L. De Raedt (ed), *Advances in Inductive Logic Programming*, pp. 82–103. IOS Press, 1996.

[61] S.H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. LNAI 1228, Springer-Verlag, 1997.

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

[63] G. Plotkin. *Automatic Methods of Inductive Inference*. Ph.D. thesis, University of Edinburgh, 1971.

[64] G. Polya. *Induction and Analogy in Mathematics* (*Mathematics and Plausible Reasoning, Volume I*). 1968. Reprinted by Princeton University Press, 1990.

[65] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning* 5:239–266, 1990.

[66] J.R. Quinlan and R.M. Cameron-Jones. FOIL: A midterm report. In P. B. Brazdil (ed), *Proc.of ECML*'93, pp. 3–20. LNCS 667, Springer-Verlag, 1993.

[67] C. Rouveirol and J.-F. Puget. Beyond inversion of resolution. In *Proc. of ICML'*90. 1990.

[68] E.Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1983.

[69] E.Y. Shapiro. Inductive inference of theories from facts. In J.-L. Lassez and G.D. Plotkin (eds), *Computational Logic: Essays in Honor of Alan Robinson*, pp. 199–254. The MIT Press, 1991.

[70] D.R. Smith. The synthesis of LISP programs from examples: A survey. In A.W. Biermann, G. Guiho, and Y. Kodratoff (eds), *Automatic Program Construction Techniques*, pp. 307–324. Macmillan, 1984.

[71] Z. Somogyi, F. Henderson, and T. Conway. Mercury: An efficient purely declarative logic programming language. In *Proc. of the Australian Computer Science Conference*, pp. 499–512, 1995.

[72] I. Stahl. Predicate invention in inductive logic programming: An overview. In P.B. Brazdil (ed), *Proc. of ECML'*93, pp. 313–322. LNAI, Springer-Verlag, 1993.

[73] I. Stahl. The appropriateness of predicate invention as bias shift operation in ILP. *Machine Learning* 20(1–2):95–117, July/Aug. 1993.

[74] L.S. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In J.-M. Jacquet (ed), *Constructing Logic Programs*, pp. 127–140. John Wiley, 1993.

[75] P.D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM* 24(1):161–175, Jan. 1977.

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

[77] N.L. Tinkham. Schema induction for logic program synthesis. *Artificial Intelligence* 98(1–2):1–47, January 1998.

[78] I. Weber. ILP systems on the ILPnet systems repository. Available as http://www-ai.ijs.si/ilpnet/irenefinal.ps, 1996.

[79] 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.

[80] S. Yılmaz. *Inductive Synthesis of Recursive Logic Programs*. M.Sc. Thesis, Ankara, Turkey, 1997. (Available as http://www.csd.uu.se/~pierref/pub/SerapMSc.ps.gz)