Completing Open Logic Programs by Constructive Induction

Esra Erdem
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712, USA
Email: esra@cs.utexas.edu

Pierre Flener
Department of Information Science
Uppsala University
Box 311, S-751 05 Uppsala, Sweden
Email: pierref@csd.uu.se

Abstract

We consider part of the problem of schema-biased inductive synthesis of recursive logic programs from incomplete specifications, such as clausal evidence (for instance, but not necessarily, ground positive and negative examples). After synthesizing the base clause and introducing recursive call(s) to the recursive clause, it remains to combine the overall result from the partial results obtained through recursion, so as to complete the recursive clause. Evidence for this combination relation can be abduced from the initially given evidence for the top-level relation. A program for this combination relation can be anything, from a single clause performing a unification (such as for *lastElem*) to multiple guarded clauses performing unifications (such as for filtering programs) to recursive programs (such as for naive *reverse*). Existing methods cannot induce guarded clause programs for this combination relation from the abduced evidence. Some existing methods cannot even detect that the combination program itself may have to be recursive and thus they then do not recursively invoke themselves the overall recursive program synthesizer. We introduce our *Program Completion Method* as a suitable extension and generalization of the existing methods.

1 Introduction

We consider part of the problem of inductive synthesis of recursive programs from incomplete specifications [9]. This is a machine learning problem, and we consider it in the logic programming framework, taking thus an ILP (Inductive Logic Programming) approach [18]. Moreover, note that we only focus on the learning of *recursive* programs, which is what we call *inductive program synthesis*. This is an important problem because necessarily invented predicates have recursive programs [20]. Yet most general-purpose ILP techniques are outperformed on this task by special-purpose recursion synthesizers [9], so that the latter seem preferable (as auxiliary tools) in case necessary predicate invention is detected or conjectured. Every now and then, inductive synthesizers appear, having the following *basic synthesis algorithm* [9], given evidence for a top-level relation r (for instance, but not necessarily, in the form of ground positive and negative examples):

1. *Schema-biased* [1] *creation* of an *open* [10] recursive program that has two clauses for r, namely a non-recursive one for a base case and a recursive one for a step case. The program is open in the sense that its recursive clause for r refers to a relation q combining the partial results (stemming from the recursive calls) into the overall results, which relation is still undefined (i.e., it has no clauses yet).

2. *Abductive generation* of evidence for q by running the open program on the evidence for r.

3. *Inductive generalization* of the abduced positive evidence and analysis of the result: if "acceptable", use it as a definition of q, thus finishing the synthesis; otherwise, conjecture necessary predicate invention [20] and recursively invoke the basic synthesis algorithm on the abduced positive and negative evidence, yielding a program for q, which, added to the initial program, provides a program for r. In any case, this amounts to closing the open program.

[1] A *schema* is a program encoding the control flow and dataflow of a class of programs (e.g., divide-and-conquer) by abstracting away their specific computations and data structures [10].

Some synthesizers of this category are THESYS [22], *BMWk* [14, 17], SYNAPSE [8, 5], LOPSTER [15], CILP [16], CRUSTACEAN [1], METAINDUCE [13], DIALOGS [7, 24], etc (see Section 5.1 for details). In order to illustrate this basic synthesis algorithm and to expose its potential weak spots, let us study a few sample runs. However, in this paper, we will almost completely ignore the mechanics of Steps 1 and 2: there are various ways of achieving the results reported hereafter (or similar ones) and we invite the reader to accept them as such, because our focus will be mostly on Step 3.

Example 1 Starting from the informal specification:

$lastElem(E, P, L)$ iff the last element of list L is E,
and list P is the corresponding prefix of L

the specifier could give the following specification by examples:

$lastElem(a, [\,], [a])$
$lastElem(b, [c], [c, b])$ $\qquad \neg lastElem(g, [h], [g, h])$
$lastElem(d, [f, e], [f, e, d])$

Suppose Step 1 creates the open program (the undefined relation is called *cons* for convenience, since it is similar to the *cons* in LISP):

$lastElem(E, [\,], [E]) \leftarrow$
$lastElem(E, [HP|TP], L) \leftarrow lastElem(E, TP, TL), cons(HP, TL, L)$

Suppose Step 2 abduces the following examples for the undefined relation:

$cons(c, [b], [c, b])$ $\qquad \neg cons(h, [g], [g, h])$
$cons(f, [e, d], [f, e, d])$

Suppose Step 3 induces the least generalization under θ-subsumption (denoted by lgθ, see Section 2.2) of the positive examples: the result $cons(A, [B|T], [A, B|T])$ is "acceptable" and can thus be unfolded into the second clause, yielding the final program:

$lastElem(E, [\,], [E]) \leftarrow$
$lastElem(E, [HP|TP], [HP, B|T]) \leftarrow lastElem(E, TP, [B|T])$

which is correct with respect to (wrt) the informal specification above. □

Example 2 Starting from the informal specification:

$reverse(L, R)$ iff list R is the reverse of list L

the specifier could give the following specification by examples:

$reverse([\,], [\,])$
$reverse([a], [a])$
$reverse([b, c], [c, b])$ $\qquad \neg reverse([g, h], [g, h])$
$reverse([d, e, f], [f, e, d])$

Suppose Step 1 creates the open program (the undefined relation is called *lastElem* for convenience):

$reverse([\,], [\,]) \leftarrow$
$reverse([HL|TL], R) \leftarrow reverse(TL, TR), lastElem(HL, TR, R)$

Suppose Step 2 abduces the following examples for the undefined relation:

$lastElem(a, [\,], [a])$
$lastElem(b, [c], [c, b])$ $\qquad \neg lastElem(g, [h], [g, h])$
$lastElem(d, [f, e], [f, e, d])$

Suppose Step 3 induces the lgθ of the positive examples: the result $lastElem(A, T, [B|V])$ is not "acceptable". Recursive invocation of the basic synthesis algorithm on the abduced examples yields the scenario of Example 1, whose final program, added to the clauses for *reverse* above, yields a final program for *reverse* that is correct wrt the informal specification above. □

So far, we have shown two successful runs of the basic synthesis algorithm, the latter featuring a recursive invocation of this algorithm. It remains however to clarify the criterion of "acceptability" of an lgθ. There are many definitions for this, and we will come back to it in Section 3. Basically, one would want the "output" terms of the lgθ to be constructed from its "input" terms: for instance, in $cons(A, [B|T], [A, B|T])$, "output" term $[A, B|T]$ *is* constructed using "input" terms A and $[B|T]$, whereas in $lastElem(A, T, [B|V])$, "output" term $[B|V]$ is *not* constructed using "input" terms A and T. Also, one would want the lgθ not to cover any abduced negative evidence for the undefined relation.

Interlude: A divide-and-conquer schema. In all runs shown here, the schema underlying Step 1 is a divide-and-conquer schema, a quite general expression of which is as follows:

$$
\begin{array}{ll}
r(X, Y, Z) \leftarrow & \\
\quad\quad solve(X, Y, Z) & \\
r(X, Y, Z) \leftarrow & \\
\quad\quad decompose(X, \overrightarrow{HX}, \overrightarrow{TX}), & \%\ \overrightarrow{HX} = HX_1, \ldots, HX_h \\
\quad\quad r(TX_1, TY_1, Z), \ldots, r(TX_t, TY_t, Z), & \%\ \overrightarrow{TX} = TX_1, \ldots, TX_t \\
\quad\quad compose(\overrightarrow{HX}, \overrightarrow{TY}, Y, Z) & \%\ \overrightarrow{TY} = TY_1, \ldots, TY_t
\end{array}
$$

Parameter X of r is the *induction parameter* (as it is decomposed for recursive calls), parameter Y is the optional *result parameter* (as it is constructed from the partial results TY_i obtained through recursion), and parameter Z is the optional *passive parameter* (as it is not decomposed for recursive calls, but serves to solve the base case and/or to compose the partial results TY_i of the step case into Y). An even more general expression of this schema would parameterize the numbers of induction, result, and passive parameters [5]. Also, in the examples above, Step 1 of the basic synthesis algorithm instantiated the h, t, r, *solve*, and *decompose* place-holders, so that the undefined relation actually was the *compose* place-holder. Using another *strategy* [10] on the same schema, Step 1 could have instantiated h, t, r, *solve*, and *compose*, so that the undefined relation actually is *decompose*. In either strategy, the instantiation of *decompose* (resp. *compose*) at Step 1 can be performed by simple re-use from a repository of "classical" such operators, the remaining *compose* (resp. *decompose*) being then instantiated at Steps 2 and 3. Since most divide-and-conquer programs have either a "classical" *decompose* or a "classical" *compose*, these two strategies cover a lot of ground. Of course, other schemas and strategies can underlie Step 1, such as the accumulator schema [6].

Let us continue now and show an unsuccessful run of the basic synthesis algorithm.

Example 3 Starting from the informal specification:

$delOddElems(L, R)$ iff list R is integer-list L without its odd elements

the specifier could give the following specification by clausal evidence (note the arrows now):

$$
\begin{array}{ll}
delOddElems([\,], [\,]) \leftarrow & \\
delOddElems([1], [\,]) \leftarrow & \quad\quad \leftarrow delOddElems([5], [5]) \\
delOddElems([2], [2]) \leftarrow & \\
delOddElems([3, 4], [4]) \leftarrow & \\
delOddElems([6, 7, 8], [6, 8]) \leftarrow &
\end{array}
$$

Suppose Step 1 creates the open program (the undefined relation is called *combine* for convenience):

$$
\begin{array}{l}
delOddElems([\,], [\,]) \leftarrow \\
delOddElems([HL|TL], R) \leftarrow delOddElems(TL, TR), combine(HL, TR, R)
\end{array}
$$

Suppose Step 2 abduces (with the help of the specifier and/or background knowledge) the following clausal evidence for the undefined relation:

$combine(1, [\,], [\,]) \leftarrow odd(1)$ $\qquad\qquad\qquad \leftarrow combine(5, [\,], [5])$
$combine(2, [\,], [2]) \leftarrow even(2)$
$combine(3, [4], [4]) \leftarrow odd(3)$
$combine(6, [8], [6, 8]) \leftarrow even(6)$

Suppose Step 3 induces the lgθ of the left-hand (positive) evidence: the result $combine(X, T, V) \leftarrow$ is not "acceptable". However, recursive invocation of the basic synthesis algorithm on all the abduced evidence will *not* yield a final program that is correct wrt the informal specification above. In fact, the *combine* relation should be defined as follows:

$combine(I, L, L) \leftarrow odd(I)$
$combine(I, L, [I|L]) \leftarrow even(I)$

Note that each of these clauses is the lgθ of some subset of the left-hand evidence. Also notice that *combine* is thus defined as the conjunction of *several* guarded clauses, with bodies involving relations other than *combine* (namely guards, or tests), rather than as a unit clause (like *cons* in Example 1) or as two clauses, one of which being recursive (like *lastElem* in Example 2). □

Objectives and organization of this paper. Basing Step 3 of the basic synthesis algorithm on the computation of the lgθ of *all* the abduced positive evidence (when it just consists of examples) thus rests on two restrictive assumptions:

1. the undefined relation is definable by a single clause;
2. the undefined relation is definable by using only unification.

The combination of these assumptions amounts to saying that the undefined relation is definable by a unit clause. However, as Example 3 shows, this is not always the case. In this paper, we will mostly address Assumption 1, namely by showing how a multi-clausal (i.e., conjunctive) definition of the undefined relation can be inductively inferred. This basically requires a re-definition of the concept of lgθ: since *unique*, over-general lgθs, such as the one in Example 3, have to be avoided, the idea is to re-define the lgθ of a clause set $\mathcal{C}$ as being a minimal-sized *set* of e clauses c_i, where $e \leq |\mathcal{C}|$, such that each c_i is the classical least generalization (under θ-subsumption) of some subset $\mathcal{C}_i$ of $\mathcal{C}$, and such that the union of the $\mathcal{C}_i$ is $\mathcal{C}$ (i.e., $\bigcup_{i=1}^{e} \mathcal{C}_i$ must form a *cover* of $\mathcal{C}$). The clauses in each $\mathcal{C}_i$ must be two by two "compatible", in the sense that they construct their result parameters in the same way. "Compatibility" of two clauses is achieved if their classical lgθ also constructs its result parameters in the same way: we approximate this by requiring that this lgθ constructs its result parameters in an "admissible" way, namely by respecting certain dataflow constraints captured in what we call a "construction mode". The generalizing clauses c_i ought to be minimal in number, because otherwise equality, for instance, would be an acceptable implementation of the re-defined operator (the chosen subsets would then all be singleton sets containing one of the clauses of $\mathcal{C}$), so that no generalization would then have been performed. Since we do not constrain the $\mathcal{C}_i$ to form a partition of $\mathcal{C}$, a clause of $\mathcal{C}$ may participate in several $\mathcal{C}_i$: this feature may favorably increase the generality of the c_i (within the bounds of admissibility) compared to an approach by simple partitioning, because the $\mathcal{C}_i$ may thus be larger; also, this feature does not increase the number of $\mathcal{C}_i$ compared to an approach by simple partitioning, because every partition is a cover and we look for a minimal cover (i.e., a cover with the smallest number of subsets). Such lgθ clauses are non-recursive if the clauses in $\mathcal{C}$ are non-recursive, but it may happen that the defined predicate does not have a correct non-recursive definition (given the current background knowledge): this is an undecidable property [20] and thus needs to be approximated by a heuristic, which we call the "acceptability" criterion.

In the rest of this paper, we will first give, in Section 2, precise meanings to the words between double quotes. Then, in Section 3, we can design a powerful new method for Step 3, called the *Program Closing Method*. It turns out that this method also lifts Assumption 2, but this requires that Step 2 provides evidence for the undefined relation that already contains all relations other than equality, or that a Step 4 be added to really close the program by adding the missing guard literals. Our method can handle clausal evidence rather than just examples. We aim at making our definitions and method as general as possible, so that they can be plugged into *any* inductive synthesizer of the considered kind, whether existing or forthcoming: therefore, independence of the schema underlying Step 1, independence of the place-holder representing the undefined relation, and independence of the mechanisms for Steps 1 and 2 will be achieved. In Section 4, we extend this method so that it can cope with non-deterministic programs. In Sections 5 and 6, we review related work and outline future work, respectively, and finally we conclude in Section 7.

2 Basic Concepts

After introducing the notation used (in Section 2.1), we define the basic concepts underlying our Program Closing Method, namely generality (in Section 2.2), construction modes (in Section 2.3), admissibility (in Section 2.4), and compatibility (in Section 2.5). This allows us to introduce our re-definition of the concept of least generalization of a clause set (in Section 2.6).

2.1 The Notation

In expressions (i.e., literals or terms) appearing in logic programs, symbols starting with uppercase letters designate (individual) variables, whereas all other symbols designate either functions or relations, the distinction (if needed) being always clear from context. All these symbols may be subscripted with natural numbers or mathematical variables (ranging over natural numbers).

When we want (or need) to group several terms into a single term, we represent this as a tuple, using angled brackets. For instance, $\langle f(a, 22), g(X, Y, Z)\rangle$ is a term representing the couple built of the two terms $f(a, 22)$ and $g(X, Y, Z)$.

When we do not want to (or cannot) fix the arity of a relation symbol, we use a "..." ellipsis notation in conjunction with subscripted variables (ranging from 1) as long-hand, and a vector notation as short-hand. For instance, atom $r(X, \vec{Y}, \vec{Z})$ is an abbreviation for $r(X, Y_1, \ldots, Y_y, Z_1, \ldots, Z_z)$, where mathematical variables y and z must be introduced in the context, and can be particularized to any natural number.

2.2 Generality

For the sake of our Program Closing Method, a very simple generality order will suffice, namely θ-subsumption [19]. Let us first repeat its definition.

Definition 1 (Term/literal generality)
A term g is *more general than* a term s if there exists a substitution σ such that $s = g\sigma$. Similarly for literals, provided they have the same sign, the same relation symbol, and the same arity.
The *least generalization* of two terms s and t, denoted by $lgt(s, t)$, is a term g that is more general than s and t, but less general than any other term u that is more general than s and t; it is always defined and is any new variable if s and t have different function symbols or different arities. Similarly for the *least generalization* of two literals a and b; it is denoted by $lgl(a, b)$ and is undefined if a and b have different signs, different relation symbols, or different arities.

For instance, term/atom $f(X, 4, Y)$ is more general than term/atom $f(2, 4, Z)$, with $\sigma = \{X/2, Y/Z\}$. The least generalization of terms/atoms $f(1, E, s, [\ ], L, [a, b])$ and $f(1, [a, b], X, [d], M, E)$ is term/atom $f(1, Q, W, T, Y, R)$. Note that Q and R are different variable symbols, even though both generalize terms E and $[a, b]$ (though in different orders): otherwise the two given terms/atoms would not be more specific than their least generalization.

We can now define a simple order of generality for clauses [19]. We assume that clauses are seen as sets of (positive and negative) literals. In this paper, we only consider definite clauses, rather than full clauses.

Definition 2 (Clause θ-subsumption)
A clause g *θ-subsumes* a clause s if there exists a substitution σ such that $g\sigma \subseteq s$.
We also say that g is *more general* than s *under θ-subsumption*.

For instance, the clause $combine(I, L, [I|L]) \leftarrow even(I)$ θ-subsumes the clause $combine(X, [HL|TL], [X, HL|TL]) \leftarrow even(X), list(TL)$ with $\sigma = \{I/X, L/[HL|TL]\}$.

When a clause g θ-subsumes a clause s, then g is more general than s, in the sense that $g \models s$. However, when g is more general than s, then g does not necessarily θ-subsume s [19]. This may happen when g and s are recursive. For instance, take g as $p(f(X)) \leftarrow p(X)$ and s as $p(f(f(X))) \leftarrow p(X)$. So θ-subsumption is only an approximation of a generality order, but a correct one, and even a sufficient one for our purposes (as we do not consider recursive clauses).

As a partial order, the θ-subsumption relation induces a lattice on the clause set, with the empty clause as the unique top element. A constructive definition of the least upper-bound-operator for this lattice emerges as a property [19]:

Property 1 (Least generalization, under θ-subsumption, of two clauses)
The *least generalization under θ-subsumption* (or lgθ) of two clauses c and d, denoted by $lg\theta(c, d)$, is the unique (up to variable renaming) clause $\{lgl(l, m) \mid l \in c \wedge m \in d\}$.

For instance, the lgθ of the clauses $combine(2, [\,], [2]) \leftarrow even(2)$ and $combine(6, [8], [6, 8]) \leftarrow even(6)$ is $combine(I, L, [I|L]) \leftarrow even(I)$.

We also need to compute the least generalization of a non-empty *set* of clauses. Again, a constructive definition of this operator emerges as a property:

Property 2 (Least generalization, under θ-subsumption, of a clause set)
The *least generalization under θ-subsumption* of a non-empty set $\mathcal{C} = \mathcal{D} \cup \{c\}$ of clauses, denoted by $lg\theta(\mathcal{C})$, is $lg\theta(lg\theta(\mathcal{D}), c)$ if $\mathcal{D}$ is non-empty, and c otherwise.[2]

This is what we here call the classical definition of this concept. One of its problems is that the *single* clause that θ-subsumes all the clauses in the set $\mathcal{C}$ is often too general, as illustrated in Example 3. In Section 2.6, we will propose a re-definition of this concept.

2.3 Construction Modes

Informally, a construction mode [3] for a relation states which parameters are constructed from which other parameters, and also expresses whether such construction is mandatory or optional. For instance, in *append*, the third parameter is mandatorily constructed from the first two parameters. Contrary to the well-known input modes (or call modes), there is no notion of different usages of a relation according to construction modes. Indeed, construction modes are a declarative notion, whereas input modes are an operational notion. However, a construction mode may not be unique for a given relation, because there may be several ways of expressing its dataflow. For instance, in *reverse*, the second parameter is mandatorily constructed from the first parameter, or vice-versa.

Definition 3 (Construction modes)
A *construction mode* m for a relation r of arity n is a total function from the set $\{1, 2, \ldots, n\}$ into the set $\{may, must, res\}$, such that res is in the range of m iff may or $must$ is in the range of m. We say that $m(i)$ is the *mode* of the i^{th} parameter of r.

A construction mode m is here written in the more suggestive form $r(m(1), \ldots, m(n))$. Since we do not consider input modes in this paper, we often simply speak about modes here.

For instance, $combine(must, must, res)$ is a construction mode, meant to express that the third parameter is a result that must be constructed from the first two parameters. Note that $p(may, must)$ is not a mode, because no parameter is being designated as the result of construction from the two given parameters. Also, $q(res)$ is not a mode, because it does not indicate from what parameters the given result parameter must or may be constructed.

The intended semantics of a construction mode is thus as follows:

- mode res means the parameter in the corresponding position is the *result parameter* to be constructed from the other parameters;
- mode $must$ means the parameter in the corresponding position is mandatory in constructing the parameter in the corresponding position of res;

[2]Note that the outer, binary occurrence of $lg\theta$ refers to the operator used in the previous property. We thus use the same name (though with different arities) for both operators, assuming that no confusion will arise.

- mode *may* means the parameter in the corresponding position is optional for constructing the parameter in the corresponding position of *res*.

We will formalize all this in the following, via the concept of admissibility. But let us first see a few other examples of construction modes.

Example 4 Let us come back to the divide-and-conquer schema of Section 1. Using general knowledge of the divide-and-conquer design methodology, it is possible to conjecture that, in general, the construction mode of $compose(\overrightarrow{HX}, \overrightarrow{TY}, Y, Z)$ is

$$compose(\overrightarrow{may}, \overrightarrow{must}, res, may)$$

where $\overrightarrow{may}$ denotes $may, \ldots, may$ with h occurrences of may, and $\overrightarrow{must}$ denotes $must, \ldots, must$ with t occurrences of $must$. (Remember that h is the number of heads HX_i, and that t is the number of tails TX_i, hence also the number of tails TY_i.)

The tails TY_i being obtained through recursion, they must all somehow be used to construct Y, because some of the recursive calls would otherwise have been useless.

The heads HX_i need not always be used to construct Y, as it all depends on the particular relation. For instance, the mode for $combine(HL, TR, R)$ (see Example 3) is $combine(must, must, res)$, the mode for $lastElem(HL, TR, R)$ (see Example 2) is $lastElem(must, must, res)$, and the mode for $cons(HP, TL, L)$ (see Example 1) is $cons(must, must, res)$. Also consider the program for $length(L, N)$ that expresses N as a Peano number and that has L as induction parameter: its instance of *compose* is $addOne(HL, TN, N)$ (defined by the clause $addOne(\_, X, s(X)) \leftarrow$) with mode $addOne(may, must, res)$. So there is no fixed mode for the heads of the induction parameter, and their most general mode thus is *may*.

The passive parameter Z also need not always be used to construct Y. For instance, for $insert(I, L, R)$, when L is the induction parameter, R the result parameter, and I the passive parameter, the instance of *compose* is $cons'(HL, TR, R, I)$ (defined by the clause $cons'(H, T, [H|T], \_) \leftarrow$) with mode $cons'(must, must, res, may)$. However, for $plateau(N, E, P)$ (which holds iff non-empty list P has exactly N elements, all equal to term E), when N is the induction parameter, P the result parameter, and E the passive parameter, the instance of *compose* is $cons''(HN, TP, P, E)$ (defined by the clause $cons''(\_, T, [H|T], H) \leftarrow$) with mode $cons''(may, must, res, must)$. So there also is no fixed mode for the passive parameter, and its most general mode thus also is *may*.

Similarly, one can argue that, in general, the construction mode of $decompose(X, \overrightarrow{HX}, \overrightarrow{TX})$ is

$$decompose(res, \overrightarrow{must}, \overrightarrow{must})$$

that the mode of $solve(X, Y, Z)$ is

$$solve(may, res, may)$$

and that the mode of $r(X, Y, Z)$ is

$$r(may, res, may)$$

All this can be even further generalized, namely for a more general divide-and-conquer schema covering arbitrary n-ary relations rather than the unary, binary, and ternary relations covered by the schema given above. One would then have a vector $\vec{X}$ of x induction parameters, a vector $\vec{Y}$ of y result parameters, and a vector $\vec{Z}$ of z passive parameters, such that $n = x + y + z \geq 1$. The resulting general modes for its place-holders become quite complicated to express (one must have recourse to tupling terms into a single term), but are beyond the scope of this paper, our objective here being merely to establish some simple concepts. □

The key issue is that construction modes can easily be pre-determined for the place-holders of any schema, no matter how complex they get, and that they can then be simply injected as arguments into the Program Closing Method described in Section 3. This means that, for the basic synthesis algorithm, the provider of evidence for the top-level relation r does *not* have to provide the mode for the open relation q, because it is already pre-computed once-and-for-all by the provider of the used schema.

Let us finish this sub-section with some useful notation. Let m be a mode for a relation r, and let $r(t_1, \ldots, t_n)$ be the head of some definite clause. Then let $Must = \langle t_i \mid m(i) = must \rangle$. Similarly for May and Res.

For instance, let the construction mode be $combine(must, must, res)$ and let the clause be $combine(6, [8], [6, 8]) \leftarrow even(6)$. We then have that $Must = \langle 6, [8] \rangle$, $May = \langle \rangle$, and $Res = \langle [6, 8] \rangle$.

2.4 Admissibility of a Clause wrt a Construction Mode

The concept of admissibility captures what it means for a definite clause to satisfy a construction mode for its relation. First, we introduce a few necessary concepts towards its definition.

Let $r(t_1, \ldots, t_n) \leftarrow \mathcal{B}$ be a definite clause, where $\mathcal{B}$ is a conjunction of atoms, called the *body* of the clause, and $r(t_1, \ldots, t_n)$ is called the *head* of the clause. It is crucial that body $\mathcal{B}$ does not contain any equality atoms, because otherwise insufficient structure would be in the parameters in the head. For instance, instead of $insert(X, [Y|L], R) \leftarrow X \leq Y, R = [X, Y|L]$, we prefer $insert(X, [Y|L], [X, Y|L]) \leftarrow X \leq Y$.

Definition 4 (Proper and reconcilable clauses)
We refer to an equality-free definite clause as a *proper clause*.
Two proper clauses are *reconcilable* if they define the same relation.

We also need some concepts for term inspection:

Definition 5 (Leaves and vertices of a term)
The *leaves* of a term t, denoted by $leaves(t)$, are the set of the variables and constants in t.
The *vertices* of a term t, denoted by $vertices(t)$, are the multi-set of the variables and functions (including the constants) in t.

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

Now, here is our (here suitably simplified) definition of admissibility [3]:

Definition 6 (Clause admissibility and clause set admissibility)
A proper clause $r(t_1, \ldots, t_n) \leftarrow \mathcal{B}$ is *admissible* wrt a mode m for r if the following conditions hold:

$$vertices(Must) \sqsubseteq vertices(\langle Res, \mathcal{B}' \rangle)$$

$$leaves(Res) \subseteq leaves(\langle May, Must, \mathcal{B}', 0, nil, \ldots \rangle)$$

where $\sqsubseteq$ denotes multi-set inclusion, and $\mathcal{B}'$ is a tuple built of the atoms (seen as terms) of $\mathcal{B}$.
A set of reconcilable clauses is *admissible* wrt a mode m for the relation in their heads if each of its clauses is admissible wrt m.

For instance, the clause $min(X, Y, X) \leftarrow X \leq Y$ is admissible wrt $min(must, must, res)$, the clause $insert(X, [Y|L], [X, Y|L]) \leftarrow X \leq Y$ is admissible wrt $insert(must, must, res)$, and the clause $addPlateau(a, [\ ], [a, s(0)]) \leftarrow$ is admissible wrt $addPlateau(may, must, res)$. However, the clause $insert(X, [Y], [X, Z]) \leftarrow$ is not admissible wrt $insert(must, must, res)$.

Note that the two conditions are more general than sub-term (i.e., sub-tree) checking. This additional generality is crucial in many cases. For instance, in atom $efface(d, [g, e, d], [g, e])$, the vertices of $[g, e]$ are a sub-multi-set of the vertices of $[g, e, d]$, but $[g, e]$ is not a sub-tree of $[g, e, d]$.

2.5 Compatibility

Intuitively, compatibility of two clauses is meant to express that these two clauses perform their result construction in the same way. Since this is a rather hard to define notion, we want to approximate it by requiring that the least generalization of these two clauses also performs the result construction in the same way. Even this is hard to capture, but it can also be approximated, namely by requiring that this least generalization not be overly general. For instance, admissibility of a proper clause wrt a construction mode for the relation in its head is such an over-generality criterion. Hence the following (here suitably simplified) definition [4]:

Definition 7 (Clause compatibility and clause set compatibility)
Two reconcilable clauses c and d are *compatible* (*with each other*) wrt a mode m for the relation in their heads if $lg\theta(c,d)$ is admissible wrt m. We also say that c is *compatible with* d wrt m, and vice-versa.
A non-empty set $\mathcal{C}$ of reconcilable clauses is *compatible* wrt a mode m for the relation in their heads if any two clauses in $\mathcal{C}$ are compatible wrt m.

For instance, the clauses $combine(2,[\ ],[2]) \leftarrow even(2)$ and $combine(6,[8],[6,8]) \leftarrow even(6)$ are compatible wrt $combine(may,must,res)$, because their lgθ, namely $combine(I,L,[I|L]) \leftarrow even(I)$, is admissible wrt that mode.

Note that a singleton set is trivially compatible wrt *any* mode. When a mode m has been clearly stated in context, we often drop the qualifier "wrt mode m", both for admissibility and for compatibility. The compatibility relation is symmetric, but it is neither reflexive nor transitive, and even should not be so [4].

2.6 A Re-Definition of the Least Generalization of a Clause Set

As argued in Section 1, the concept of least generalization of a clause set ought to be re-defined such that it yields a minimal-sized *set* of clauses (rather than a single clause), each of which being not too general. We have by now precisely defined all the vague concepts (namely construction mode, admissibility, and compatibility) at the end of the introduction, so that we are now able to propose our re-definition, such that it fits all our requirements:

Definition 8 (Least generalization, under θ-subsumption, of a clause set, wrt a construction mode)
The *least generalization under θ-subsumption* of a non-empty set $\mathcal{C}$ of clauses wrt a construction mode m, denoted by $lg\theta(\mathcal{C},m)$, is the set of classical least generalizations under θ-subsumption of the node cliques of the graph with node set $\mathcal{C}$ and edge set induced by the compatibility relation wrt m.

For instance, the least generalization wrt $combine(must,must,res)$ of the four (left-hand) *combine* clauses in Example 3 is indeed made of the two clauses mentioned in that example.

We have elsewhere given an efficient approximation algorithm solving this NP-complete problem (for any over-generality criterion) [4].

3 The Program Closing Method

Given:

- an open logic program $\mathcal{R}$ for a relation r, with no clauses for some relation q,
- a set $\mathcal{E}$ of reconcilable non-recursive clauses for q, called the *evidence set*,
- a construction mode m for q,

the objective of the *Program Closing Method* is to infer a closed program $\mathcal{R}' = \mathcal{R} \cup \mathcal{Q}$ for r, where $\mathcal{Q}$ is a logic program for q that is more general than $\mathcal{E}$ (in the sense that $\mathcal{Q} \models \mathcal{E}$).

Of course, we do not want $\mathcal{Q}$ to be equal to $\mathcal{E}$, nor to cover all syntactically possible atoms for q. What is wanted is rather that $\mathcal{Q}$ covers an "extension" of $\mathcal{E}$, such that this "extension" coincides with the unknown intended relation q.

This problem statement is not quite the same as the one of the general ILP task (where $\mathcal{R}$ is empty), because it may not "solve" the ILP problem, nor the disjunctive predicate learning problem, by itself, but only as an auxiliary to more powerful and intricate techniques. We here only target highly constrained situations, where the open relation is of the kind where some parameter is indeed mandatorily constructed from some parameters and optionally from others, and such that the corresponding construction mode is known in advance (say by pre-determination based on an underlying program schema).

The first step of the Program Closing Method is to assume that $\mathcal{Q} = lg\theta(\mathcal{E}^+, m)$, where $\mathcal{E}^+$ is the positive evidence of $\mathcal{E}$.

For instance, from the positive *cons* evidence in Example 1 and the mode $cons(must, must, res)$, this yields the single clause $cons(A, [B|T], [A, B|T]) \leftarrow$ as least generalization, which is acceptable (as it gives rise to a partially correct and complete program for *lastElem*). Also, from the positive *combine* evidence in Example 3 and the mode $combine(must, must, res)$, this yields the two clauses $combine(I, L, L) \leftarrow odd(I)$ and $combine(I, L, [I|L]) \leftarrow even(I)$ as least generalization, which is also acceptable. But, from the positive *lastElem* evidence in Example 2 and the mode $lastElem(must, must, res)$, this yields the three clauses $lastElem(a, [\ ], [a]) \leftarrow$, $lastElem(b, [c], [c, b]) \leftarrow$, and $lastElem(d, [f, e], [f, e, d]) \leftarrow$, i.e., the very input evidence, and this is not acceptable, because adding these clauses to the *reverse* ones will not yield a complete program for *reverse* (though a partially correct one).

Hence, the initial assumption is not always acceptable. Indeed, sometimes it is necessary to reject it and instead invoke an entire new synthesis of a recursive program from the evidence for q. Note that the given abduced evidence is always non-recursive, and also note that the least generalization (under θ-subsumption) of non-recursive clauses cannot be recursive, even for our re-definition of this concept. Rejection plus auxiliary synthesis corresponds to *necessary predicate invention*, 'necessary' in the sense that a recursive program $\mathcal{Q}$ for q cannot be eliminated by unfolding for occurrences of q in $\mathcal{R}$ (inducing such a $\mathcal{Q}$ is also called *constructive induction*). If the result $\mathcal{Q}$ is deemed acceptable, then relation/predicate q is 'unnecessary' in the sense that the non-recursive program $\mathcal{Q}$ can be eliminated by unfolding for occurrences of q in $\mathcal{R}$. Now, it is in general undecidable whether predicate invention is necessary (via rejection) or not (via acceptance) [20]. So a heuristic is needed to judge the output of this first computation of the Program Closing Method, and we call this heuristic the *acceptability criterion*.

Our proposed acceptability criterion is as follows. The program $\mathcal{Q}$ induced so far is *acceptable* if the following conditions hold:

1. program $\mathcal{Q}$ does not cover any of the negative evidence for q;

2. the number of clauses in $\mathcal{Q}$ is "not too large."

The first condition is obvious, as it avoids over-generalization. (It could by the way be pushed into the compatibility criterion, but that would mean much more non-coverage checks than the proposed unique final check.) Note that both $lg\theta(\mathcal{E}^+, m)$ and the acceptability criterion can be evaluated in the absence of negative evidence. The proposed work is thus suitable for the current trend on learning from positive evidence only, as negative evidence is hard to come by in some application settings. The second condition needs to be refined for each particular synthesis technique. For instance, if only carefully chosen evidence is presented to it, say e clauses, then "not too large" could mean "less than $e \div 2$". An alternative way of expressing this idea is to require that no clique has only one element (remember that each clause of $\mathcal{Q}$ is the lgθ of some clique).

Example 5 Let $\mathcal{E}$ be the following evidence set for *insert*:

$$
\begin{array}{ll}
insert(1, [\ ], [1]) \leftarrow & (E_1) \\
insert(3, [4], [3, 4]) \leftarrow & (E_2) \\
insert(4, [2], [2, 4]) \leftarrow & (E_3) \\
insert(6, [5, 7], [5, 6, 7]) \leftarrow & (E_4) \\
insert(5, [1, 3], [1, 3, 5]) \leftarrow & (E_5) \\
insert(7, [3, 6, 8], [3, 6, 7, 8]) \leftarrow & (E_6)
\end{array}
$$

and let the construction mode be $insert(must, must, res)$. Then we get the following three cliques: $\mathcal{E}_1 = \{E_1, E_2\}$, $\mathcal{E}_2 = \{E_3, E_4\}$, and $\mathcal{E}_3 = \{E_5, E_6\}$, and their least generalization is:

$$
\begin{array}{l}
insert(X, L, [X|L]) \leftarrow \\
insert(X, [Y|L], [Y, X|L]) \leftarrow \\
insert(X, [Y, Z|L], [Y, Z, X|L]) \leftarrow
\end{array}
$$

But this program does not satisfy our acceptability criterion, because the number of clauses is not less than 3 (which is half the size of the evidence set). In this program, the i^{th} clause achieves insertion of a number into the i^{th} position of a list of numbers, but this program is not as general as we want because it does not cover insertion of a number into the i^{th} position of a list of numbers where $i > 3$. This cannot be done with a finite non-recursive program (unless suitable relations are available), so the initial assumption is inadequate and detecting this is what the acceptability criterion is for. □

The second step of the Program Closing Method is thus to keep the program $\mathcal{Q}$ of the first step if the acceptability criterion is satisfied, and otherwise to recompute it as $\mathcal{Q} = ISRLP(\mathcal{E})$, where $ISRLP$ is some inductive synthesizer of recursive logic programs (which can of course be the same as the one that gave rise to the open program that needed completion in the first place).

4 The Case of Non-Deterministic Programs

The (positive) evidence abduced by Step 2 of the basic synthesis algorithm is not always in the form of proper clauses, and the Program Closing Method is then inapplicable as it stands. For the divide-and-conquer schema, such is the case when the top-level relation r is non-deterministic given particular values for the chosen induction parameter and passive parameter (if any): the recursive call to the top-level relation for a tail of the induction parameter (obtained through *decompose*) may then yield (upon backtracking) several values for the chosen result parameter, but only one of them is actually adequate to construct (through *compose*) a result corresponding to the un-decomposed induction parameter. Several such values lead to a disjunction of abduced evidence, or to overly general abduced evidence with variables instead of more specific terms in the head.

Example 6 Consider the following specification:

$perm(L, P)$ iff list P is a permutation of list L

If Step 1 generates the open program (using L as the induction parameter):

$perm(L, P) \leftarrow L = [\,], P = [\,]$
$perm(L, P) \leftarrow L = [HL|TL], perm(TL, TP), compose_1(HL, TP, P)$

then, from the evidence $perm([a, b, c], [c, a, b]) \leftarrow$, at best the following *disjunctive* evidence for $compose_1$ could be abduced by Step 2:

$compose_1(a, [b, c], [c, a, b]) \leftarrow$
$\vee\ compose_1(a, [c, b], [c, a, b]) \leftarrow$

because there are two correct instances of the recursive call $perm([b, c], TP)$, namely $perm([b, c], [b, c])$ and $perm([b, c], [c, b])$. But only the latter clause is actually adequate for constructing a program for $compose_1$ so that the query $\leftarrow perm([a, b, c], [c, a, b])$ can be proved, because if the former were chosen, then $compose_1$ would involve the same problem as *perm* itself, namely permuting a list. □

Example 7 Now consider the following specification:

$length(L, N)$ iff list L has N elements

If Step 1 generates the open program (using N as the induction parameter):

$length(L, N) \leftarrow N = 0, L = [\,]$
$length(L, N) \leftarrow N > 0, TN\ is\ N - 1, HN = \_\,, length(TL, TN), compose_2(HN, TL, L)$

then, from the evidence $length([a, b, c], 3) \leftarrow$, at best the following *over-general* evidence for $compose_2$ could be abduced by Step 2:

$compose_2(A, [B, C], [a, b, c]) \leftarrow$

because $length([B, C], 2)$ summarizes all the instances of the recursive call $length(TL, 2)$. But only one instance of this clause, namely $compose_2(a, [b, c], [a, b, c]) \leftarrow$, is actually adequate for constructing a program for $compose_2$ so that the query $\leftarrow length([a, b, c], 3)$ can be proved. □

Example 8 Finally, consider the following specification:

$prefix(L, P)$ iff list P is a prefix of list L

If Step 1 generates the open program (using L as the induction parameter):

$prefix(L, P) \leftarrow L = \_ , P = [\,]$
$prefix(L, P) \leftarrow L = [HL|TL], prefix(TL, TP), compose_3(HL, TP, P)$

then, from the evidence $prefix([a, b, c], [a, b]) \leftarrow$, at best the following *disjunctive* evidence for $compose_3$ could be abduced by Step 2:

$compose_3(a, [\,], [a, b]) \leftarrow$
$\vee\ compose_3(a, [b], [a, b]) \leftarrow$
$\vee\ compose_3(a, [b, c], [a, b]) \leftarrow$

because there are three correct instances of the recursive call $prefix([b, c], TP)$, namely $prefix([b, c], [\,])$, $prefix([b, c], [b])$, and $prefix([b, c], [b, c])$. But only the second clause is actually adequate for constructing a program for $compose_3$ so that the query $\leftarrow prefix([a, b, c], [a, b])$ can be proved, because if one of the other clauses were chosen, then $compose_3$ would have to do strange things. Indeed, notice that only the second clause is admissible wrt mode $compose_3(may, must, res)$. □

Essentially, in all these examples, a *set* of evidence clauses for the open relation q is abduced for each piece of evidence for the top-level relation r. Overall, a set of sets of clauses is thus abduced. In order for the clique cover algorithm to be applicable, each of these abduced sets needs to be trimmed down to a *single* clause, by choosing a particular clause in the set and by instantiating, if necessary, its variables. In order for this trimming to be maximally constrained (note that there is an infinity of instantiations of a clause), we require the resulting clause to be admissible wrt the mode of the open relation, which is why we call it an admissible alternative. Note that the clique cover algorithm itself does *not* require the input clauses to be admissible, because such a pre-condition is not necessary to its functioning (remember that it just uses the admissibility criterion in order to find compatible subsets). However, the admissibility criterion turns out to be very useful here as well, since, in all our practical experiments so far, it renders the number of such admissible alternatives *finite*. We conjecture that this is (almost) always the case. Our solution presented below only works when such finiteness is the case, since it iterates over all combinations of admissible alternatives. Clearly, the definition of admissibility is crucial here: the more restrictive it is, i.e., the fewer clauses it declares admissible, the fewer admissible alternatives a given clause set will have, and the fewer combinations have to be examined. Hence the following definition:

Definition 9 (Admissible alternatives of a clause set)
Clause A is an *admissible alternative* of clause set $\mathcal{C}$ wrt mode m if A is a most general instance of some element of $\mathcal{C}$ and if A is admissible wrt m.

For instance, in Example 6, both $compose_1$ clauses are admissible alternatives of the abduced clause set. In Example 7, the admissible alternatives of the abduced clause set are $compose_2(a, [b, c], [a, b, c]) \leftarrow$, $compose_2(a, [c, b], [a, b, c]) \leftarrow$, $compose_2(b, [a, c], [a, b, c]) \leftarrow$, $compose_2(b, [c, a], [a, b, c]) \leftarrow$, $compose_2(c, [a, b], [a, b, c]) \leftarrow$, and $compose_2(c, [b, a], [a, b, c]) \leftarrow$. In Example 8, only the second $compose_3$ clause is an admissible alternative of the abduced clause set. Notice thus the non-determinism of the concept.

Generating admissible alternatives for a given clause set thus amounts to first selecting one of its clauses and then, if this is possible, minimally specializing it, through instantiation of its variables, such that it becomes admissible, according to the two (multi-)set inclusion constraints in Definition 6. Judging from the examples above, this may look like quite a formidable task, both in programming effort and in having it executed quickly. Fortunately, both aspects of this task can be very elegantly addressed by using a set constraint (logic) programming language, such as COJUNTO [11], because the formulation of the task in such a language is virtually identical to the formulation given above (due to the very high expressiveness of these languages), and yet it is extremely efficient (due to the powerful underlying constraint solvers, which avoid a simple generate-and-test approach). Tighter constraints in Definition 6 may reduce the number of admissible alternatives in a problem-independent way. For instance, type constraints could be added, thus preventing for instance the instantiation of a variable of type *List* to a number.

Now, the key insight towards a generalized Program Closing Method is as follows: if, for some abduced clause sets, inadequate admissible alternatives are chosen, then there will be few compatibilities among them, hence small cliques, and therefore many cliques; conversely, if there are few cliques, then these cliques are large, hence there are many compatibilities among the clauses, and therefore all these clauses are adequate admissible alternatives. The idea is thus to iterate over the $lg\theta$ algorithm for each of the finitely many combinations of admissible alternatives, one for each abduced clause set, and to select the clique cover with the fewest cliques so as to evaluate it for acceptability. Hence the following algorithm:

Algorithm *Program Closing Method*
Inputs:
– an open logic program $\mathcal{R}$ for a relation r, with no clauses for some relation q
– a set $\mathcal{E}$ of sets $\mathcal{E}_i$ of reconcilable non-recursive clauses for q
– a construction mode m for q.
Output:
– a closed program $\mathcal{R}' = \mathcal{R} \cup \mathcal{Q}$ for r,
where $\mathcal{Q}$ is a logic program for q induced from evidence $\mathcal{E}$ wrt mode m.
initialize the current minimal cover size: $M \leftarrow \infty$;
while $M > 1$ **and** there is an uninvestigated set $\mathcal{S}_j$ of admissible alternatives, one from each $\mathcal{E}_i$, **do**
  begin
    compute a candidate solution: $\mathcal{G}_j \leftarrow lg\theta(\mathcal{S}_j, m)$;
    if $|\mathcal{G}_j| < M$ **then**
      begin {a better minimal cover is found}
        store the currently best minimal cover size: $M \leftarrow |\mathcal{G}_j|$;
        store the currently best set of admissible alternatives: $\mathcal{M} \leftarrow \mathcal{S}_j$;
        store the currently best solution: $\mathcal{Q} \leftarrow \mathcal{G}_j$
      end
  end ;
if $\neg$ *acceptable*($\mathcal{Q}$) **then**
  start a new synthesis from the best set of admissible alternatives: $\mathcal{Q} \leftarrow ISRLP(\mathcal{M})$;
return $\mathcal{R} \cup \mathcal{Q}$

Let e be the number of $\mathcal{E}_i$, and let a be the average number of admissible alternatives of the $\mathcal{E}_i$ wrt m. Then the time complexity of this algorithm is exponential in e, namely a^e, which is clearly expensive when e gets large (even though our clique cover algorithm is extremely fast, as reported in [4]). We thus recommend using this algorithm only when e is small.

In terms of experimentation with this Program Closing Method, we can report on very satisfactory results (based on a DIALOGS-like approach [7] to Steps 1 and 2). Indeed, if given minimally sufficient (even to a human synthesizer) top-level evidence, this Program Closing Method can assist with the successful synthesis of programs for all (and similar) relations mentioned in the examples of this paper.

Example 9 Take the following evidence for *length* (see Example 7):

$$length([\,], 0) \leftarrow$$
$$length([d], 1) \leftarrow$$
$$length([e, f], 2) \leftarrow$$
$$length([a, b, c], 3) \leftarrow$$

Suppose Step 1 somehow generates the open program $\mathcal{R}$:

$$length(L, N) \leftarrow N = 0, L = [\,]$$
$$length(L, N) \leftarrow N > 0, TN\ is\ N - 1, HN = \_\,, length(TL, TN), compose_2(HN, TL, L)$$

and Step 2 somehow abduces the following set $\mathcal{E}$ of evidence sets for *compose*$_2$:

$$\{\mathcal{E}_1 : \{compose_2(D, [\,], [d]) \leftarrow\},$$
$$\mathcal{E}_2 : \{compose_2(E, [F], [e, f]) \leftarrow\},$$
$$\mathcal{E}_3 : \{compose_2(A, [B, C], [a, b, c]) \leftarrow\}\}$$

By the chosen divide-and-conquer schema (see Section 1 and Example 4), the construction mode m is $compose_2(may, must, res)$. Initially, the size M of the current minimal cover is ∞. At each iteration j, one admissible alternative each from $\mathcal{E}_1$ through $\mathcal{E}_3$ is picked to form a candidate set $\mathcal{S}_j$. In this case, there are maximum $1 \times 2 \times 6 = 12$ iterations, because there are 1 (respectively 2 and 6) admissible alternatives of $\mathcal{E}_1$ (respectively $\mathcal{E}_2$ and $\mathcal{E}_3$) (see above for those of $\mathcal{E}_3$). At the first iteration, suppose the candidate set $\mathcal{S}_1$ is:

$$compose_2(d, [\,], [d]) \leftarrow$$
$$compose_2(f, [e], [e, f]) \leftarrow$$
$$compose_2(a, [b, c], [a, b, c]) \leftarrow$$

The first candidate solution $\mathcal{G}_1$ is thus the $lg\theta$ of $\mathcal{S}_1$ wrt the mode above, namely:

$$compose_2(H, T, [H|T]) \leftarrow$$
$$compose_2(f, [e], [e, f]) \leftarrow$$

As $|\mathcal{G}_1| = 2 < M$, a better minimal cover is found, so M is set to 2 and $\mathcal{Q}$ is set to the currently best solution $\mathcal{G}_1$. At the second iteration, suppose the candidate set $\mathcal{S}_2$ is:

$$compose_2(d, [\,], [d]) \leftarrow$$
$$compose_2(e, [f], [e, f]) \leftarrow$$
$$compose_2(a, [b, c], [a, b, c]) \leftarrow$$

The second candidate solution $\mathcal{G}_2$ is thus the $lg\theta$ of $\mathcal{S}_2$ wrt the mode above, namely:

$$compose_2(H, T, [H|T]) \leftarrow$$

As $|\mathcal{G}_2| = 1 < M$, a better minimal cover is found, so M is now set to 1 and $\mathcal{Q}$ is now set to the currently best solution $\mathcal{G}_2$. As the size of this cover is 1, and as there cannot be a cover of size less than 1, the loop produces $\mathcal{G}_2$ as the best solution $\mathcal{Q}$. Since it is also acceptable (because its number of clauses is less than half the number of elements in $\mathcal{E}$), the set $\mathcal{R} \cup \mathcal{Q}$ is returned, which is indeed a totally correct program for *length*. □

5 Related Work

There are two kinds of related work. First, in Section 5.1, we review inductive synthesis techniques that more or less follow the basic synthesis algorithm of Section 1, by showing how they deviate from that algorithm as well as in what sense the Program Closing Method presented here generalizes the corresponding methods in these techniques, and could thus be plugged into them to increase their power (i.e., the size of the class of relations for which programs can be successfully synthesized), if not to correct their flaws. Then, in Section 5.2, we compare our Program Closing Method to other methods, which have been proposed independently of particular synthesis techniques.

5.1 Related Synthesis Techniques

This paper is a considerable extension of the second author's previous work on the SYNAPSE synthesis technique [8, 5], and is based on the advances reported by the first author [2]. SYNAPSE features a slight variation of the basic synthesis algorithm, and starts from positive examples as well as properties (expressed by non-recursive definite clauses) as specification of the top-level relation. It is biased by a (hardwired) divide-and-conquer schema, which is however more informative than the one in Section 1, in the sense that the *compose* place-holder is split into a conjunction of two place-holders, namely *processCompose* for combining partial results into overall results, and *discriminate* for discriminating between alternative instances of the former. The key difference with the basic synthesis algorithm is that its Step 2 only abduces examples of *processCompose*, so that a Step 4 needs to be added to abduce the instances of *discriminate*, which is done by a

Proofs-as-Programs Method, using the properties. The program closing method of SYNAPSE, called the *MSG Method*, is a precursor to the version presented here, in the sense that the definition of admissibility is considerably more powerful now: less evidence is now considered admissible, and we now also handle clausal evidence, and hence forms of semantic construction. The definitions of construction mode and compatibility, as well as the algorithm for computing least generalizations, have also undergone significant improvements.

The DIALOGS synthesis technique [7] now [24] exploits many advances presented here. It also features a slight variation of the basic synthesis algorithm, as it starts from no specification at all and collects its (positive) evidence by querying the specifier. The key difference with the basic synthesis algorithm is that its Step 1 does not instantiate the *solve* place-holder, so that its Step 2 simultaneously abduces evidence of *solve* and of *compose*, and that its Step 3 decides which pieces of this evidence are used to instantiate which of these place-holders. Non-deterministic relations cannot be handled, but this can easily be achieved by adding the remaining advances reported here.

The METAINDUCE synthesis technique [13] exactly follows the basic synthesis algorithm, using the divide-and-conquer schema of Section 1, and starts from positive and negative examples of the top-level relation. Examples 1 to 3 can be acted out by this technique, including the erroneous decision about the lgθ of Example 3, due to the absence of the concepts of construction mode, admissibility, and compatibility. Its acceptability criterion simply is that the (unique) lgθ should not cover any negative evidence and that the variables of the *res* parameter are a subset of the variables of the two *must* parameters. There is thus no concept of *may* parameters, and the non-consideration of constants and functors sometimes leads to wrong decisions. Non-deterministic relations cannot be handled.

The CILP synthesis technique [16] follows the basic synthesis algorithm, except that its Step 3 is based on the concept of sub-unification, rather than anti-unification, and that it is its Step 3 that instantiates the *solve* place-holder, rather than its Step 1. The underlying schema is less informative than the one in Section 1, in the sense that it has fewer place-holders and does not prescribe the dataflow; therefore, a heuristic analysis (based on input-mode declarations) needs to be done to figure out the necessary parameters of the *compose* place-holder, instead of precompiling this once and for all at the schema-level with more precise constraints on the dataflow. The technique cannot induce multi-clausal instances of *compose*, and its acceptability criterion reduces to the over-generalization check (by rejecting programs that cover some negative evidence) (of course, it is sub-unification that allowed many simplifications of this criterion). Non-deterministic relations cannot be handled.

The CRUSTACEAN synthesis technique [1] is a successor of the LOPSTER technique [15], in the sense that a few features have been improved. However, it cannot perform necessary predicate invention, so that its Step 3 never calls CRUSTACEAN recursively. CRUSTACEAN is basically a predecessor of CILP and thus inherits the drawbacks of CILP.

The THESYS synthesis technique [22] is a precursor to all these techniques, but it is set in the functional programming paradigm (and non-deterministic relations can thus not be handled). In case of an unacceptable lgθ at its Step 3, it does not call itself recursively for the necessary predicate invention, but rather tries to *avoid* this by generalizing the given examples and re-trying from scratch (also see [6]). For instance, THESYS cannot infer a functional program for *reverse* corresponding to the naive (quadratic) *reverse* program of Example 2, but instead infers a non-naive (linear) *reverse* program based on difference lists (i.e., based on the introduction of an accumulator parameter). However, such an accumulator introduction is not always possible; for instance, in the absence of background knowledge, synthesizing a *product* functional/relational program leads to the necessary invention of a *sum* function/relation, which cannot be avoided through generalization of *product*. THESYS was the first schema-biased inductive synthesizer, and it has been extended, revised, and reformulated over the years as the *BMWk* technique [14, 17], and was also transposed to a higher-order logic framework [12].

Many other techniques of inductive synthesis of recursive programs, although they are not all schema-biased, are reviewed in [9].

5.2 Related Methods

The SIERES learning technique [23] is not really schema-biased and thus does not really follow the basic synthesis algorithm. However, it features a few components not unlike our Program Closing Method and its conceptual apparatus. Indeed, it also computes the lgθ of evidence (which must however be unit clauses); it constructs clauses that fit argument dependency graphs (a kind of primitive schemas that prescribe the dataflow but not the control flow); and it uses input-mode declarations to guide this construction towards non-over-general clauses. However, there is no notion of compatibility, and hence no possibility of division of the evidence into cliques, i.e., no inducability of multi-clausally defined relations.

The INDICO learning technique [21] is not at all an instance of the basic synthesis algorithm. However, it features an interesting method for conjecturing the heads of possible clauses, hence providing already much of the discriminating information that otherwise has to be discovered together with the characterizing information when starting from most-general clause heads. The method first partitions (i.e., it does not divide) the evidence (which must be unit clauses) into subsets according to the functors (e.g., type constructors) appearing in it; then it computes the lgθ of each obtained subset so as to produce a series of clause heads, from which a top-down clause specialization process can then be started. This method is obviously related to, but more specialized than, our clique finding mechanism.

6 Future Work

The Program Closing Method presented here is already very powerful (as it generalizes and corrects all "competing" methods known to the authors), but it can nevertheless be extended in various ways, which we examine now. We have already mentioned the existence of a more general definition of construction modes (and hence of admissibility and compatibility) in [3, 4] (where we even discuss enhancements thereof), so that more powerful recursive schemas can be supported.

Recursive evidence. As of now, the Program Closing Method is restricted to abduced evidence in the form of non-recursive (reconcilable) clauses. There is no theoretical obstacle to also allowing recursive clauses as evidence (except for the mentioned inadequacy of computing the least generalization under θ-subsumption of two recursive clauses). In fact, our restriction to non-recursive clauses was rather motivated by a pragmatic choice: if the abduced evidence were recursive, then the evidence for the top-level relation would most likely also have been recursive; but that would in turn mean that the specifier would have to provide such recursive evidence; but it seems (to us) that doing so is tantamount to already writing the program itself and that the specifier would then most likely not need an inductive synthesizer to write the program.

Number of undefined relations. The basic synthesis algorithm assumes there is only one undefined relation by the time Step 2 is reached, hence that the top-level relation can be defined in terms of a chain (rather than a tree) of invented predicates. (Note that, upon recursive invocation of the basic synthesis algorithm, a different schema can be selected at each level.) However, such is not always the case, as shown by the approach of DIALOGS [7, 24]. It would thus be interesting to investigate in full generality how to adapt the Program Closing Method when its evidence is about multiple undefined relations.

Background knowledge. An almost certain criticism of our work is that we compute generalizations in the absence of background knowledge (not to mention our usage of the "old-fashioned" θ-subsumption order for generality). However, note that we assume that the abduced evidence for the undefined relation already contains all the necessary relations, so that the responsibility of discovering them does not lie with the Program Closing Method, but with its clients, whether they achieve this by interaction with an oracle (as in DIALOGS [7, 24]), or by extraction from the evidence for the top-level relation (as in SYNAPSE [8, 5]), or by some form of background knowledge usage (as in the vast majority of inductive synthesizers). (Also remember that θ-subsumption suffices for non-recursive clauses, which are sufficient here, as argued earlier.) So our choices are rather justified, but one can of course investigate the use of background knowledge and/or a stronger order of generality in order to push the mentioned assumption inside the Program Closing Method.

7 Conclusion

We have considered part of the problem of schema-biased inductive synthesis of recursive logic programs from incomplete specifications, such as clausal evidence. The techniques that follow the outlined basic synthesis algorithm usually have a problem with their final step, which is the synthesis of a program for the relation combining the overall result from the partial results obtained through recursion. Evidence for this combination relation can be abduced from the initially given evidence for the top-level relation. A program for this combination relation can be anything, from a single clause performing a unification (such as for *lastElem*) to multiple guarded clauses performing unifications (such as for filtering programs) to recursive programs (such as for naive *reverse*). Existing methods cannot induce guarded clause programs for this combination relation from the abduced evidence. Some existing methods cannot even detect that the combination program itself may have to be recursive and thus they then do not invoke some recursion synthesizer (say themselves).

We have introduced our Program Completion Method as a suitable extension and correction of the existing methods. It is based on our re-definition of the concept of least generalization of a clause set [4], namely that it is itself a *set* of clauses (rather than a single clause), each such clause being the classical least generalization of a subset of the given clause set. Membership of a clause in such a subset is subject to its being compatible with all the other clauses, compatibility meaning that the classical least generalization does not become too general according to some over-generality criterion. Since we are here in a highly constrained situation where the combination relation is known in advance to have a certain dataflow between its parameters, we have chosen admissibility wrt a construction mode as a suitable over-generality criterion [3]. Basically, a construction mode for a relation states which parameters are constructed from which other parameters, also expressing whether such construction is mandatory or optional. For any recursive program schema, the construction mode of the relation combining the overall result from the partial results obtained through recursion can be pre-determined, at the schema level, so that it will be suitable for all particular programs fitting that schema. Our approach has the advantage of also working in the absence of negative evidence, so that over-generality is not only measured in terms of non-coverage of such negative evidence.

Acknowledgments

We thank Baudouin Le Charlier and Pierre-Yves Schobbens (both at the University of Namur, Belgium), Necip Fazıl Ayan (Ankara, Turkey), and the Machine Learning Group at the University of Texas at Austin, for their helpful suggestions on a preliminary version 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] E. Erdem. *An MSG Method for Inductive Logic Program Synthesis.* Senior Project Final Report, Ankara (Turkey), May 1996.

[3] E. Erdem and P. Flener. *A New Declarative Bias for ILP: Construction Modes.* In preparation.

[4] E. Erdem and P. Flener. *A New Heuristic to Use Least Generalizations in ILP.* In preparation.

[5] P. Flener. *Logic Program Synthesis from Incomplete Information.* Kluwer, 1995.

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

[7] P. Flener. Inductive Logic Program Synthesis with DIALOGS. In S. Muggleton (ed), *Proc. of ILP'96*, pp. 175–198. *LNAI* 1314, Springer-Verlag, 1997.

[8] P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *J. of Symbolic Computation* 15(5–6):775–805, May/June 1993.

[9] P. Flener and S. Yılmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. Accepted for publication in *J. of Logic Programming*.

[10] P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In N.E. Fuchs (ed), *Proc. of LOPSTR'97*, pp. 128–147. *LNCS* 1463, Springer-Verlag, 1998.

[11] C. Gervet. Cojunto: Constraint logic programming with finite set domains. In M. Bruynooghe (ed), *Proc. of ILPS'94*, pp. 339–358. The MIT Press, 1994.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] S. Lapointe, C.X. 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.

[17] G. Le Blanc. BMWk 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.

[18] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *J. of Logic Programming* 19–20:629–679, May/July 1994.

[19] G.D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie (eds), *Machine Intelligence* 5:153-163. Edinburgh University Press, Edinburgh (UK), 1970.

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

[21] I. Stahl, B. Tausend, and R. Wirth. Two methods for improving inductive logic programming systems. In P. Brazdil (ed), *Proc. of ECML'93*, pp. 41–55. *LNAI* 667, Springer-Verlag, 1993.

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

[23] 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.

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