A Redefinition of Least Generalizations and its Application to Inductive Logic Program Synthesis

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 Computer Engineering and Information Science
Bilkent University, 06533 Bilkent, Ankara, Turkey
Email: pf@cs.bilkent.edu.tr

Abstract

The 'classical' definition of the concept of least generalization (under θ-subsumption) of a clause set $\mathcal{C}$ is that it is a *single* clause. Since such a unique clause is sometimes over-general, we re-define this concept as being a minimal-sized *set* of clauses, each member of this set being the least generalization (under 'classical' θ-subsumption) of some subset of $\mathcal{C}$. The elements of these subsets are two by two compatible, in the sense that their least generalizations (under 'classical' θ-subsumption) are not too general. We show an algorithm for computing this redefined concept.

The criterion for over-generality is of course problem-specific. We design such a criterion for a problem frequently occurring in the inductive synthesis of recursive logic programs, namely the closing of an open program that was synthesized in a schema-biased way, but that has one of the place-holders of the used schema still undefined. After evidence for this undefined relation has been abduced, it needs to be inductively generalized. A least generalization is over-general if it is not admissible wrt a construction mode capturing the required dataflow of the place-holder. We design a language for expressing such construction modes (for any place-holder of any schema), and we define powerful admissibility and compatibility criteria. We also prove a few theorems relating the problem-independent concept of compatibility with the problem-specific concept of admissibility: these theorems show how to speed up certain computations for this specific problem.

1 Introduction

We consider (part of) the problem of inductive synthesis of recursive programs from incomplete specifications [7]. This is a machine learning problem, and we consider it in the logic programming framework, taking thus an ILP (Inductive Logic Programming) approach [17]. Moreover, note that we only focus on the learning of *recursive* programs, which is what we call *inductive program synthesis*. Every now and then, inductive synthesizers appear, having the following *basic synthesis algorithm* [7], 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 [15] 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

[1] A *schema* is a program encoding the control-flow and data-flow of a class of programs (e.g. divide-and-conquer) by abstracting away their specific computations and data structures [3].

(stemming from the recursive calls) into the overall results, which relation is still undefined (i.e. has no clauses yet).

2. *Abductive generation* of evidence for q by execution of 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 definition of q, thus completing the synthesis; otherwise, conjecture necessary predicate invention [19] and recursively invoke the basic synthesis algorithm on the entire abduced 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.

Some synthesizers of this category are THESYS [21], *BMWk* [11, 16], SYNAPSE [6, 3], LOPSTER [13], CILP [14], CRUSTACEAN [1], METAINDUCE [10], DIALOGS [5, 23], etc (see Section 4.1 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:

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

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

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

Step 2 abduces the following evidence for the undefined relation:

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

Step 3 induces $cons(A, [B|T], [A, B|T])$ as least generalization under θ-subsumption (denoted by lgθ, see Section 2.2), of the positive evidence, which 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:

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

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

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

Step 2 abduces the following evidence for the undefined relation:

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

Step 3 induces $lastElem(A, T, [B|V])$ as lgθ of the positive evidence, which is not "acceptable." Recursive invocation of the basic synthesis algorithm on the abduced evidence 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 executions 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 3.2. Basically, one would want the "output" parameters of the lgθ to be constructed from its "input" parameters: for instance, in $cons(A, [B|T], [A, B|T])$, parameter $[A, B|T]$ *is* constructed using parameters A and $[B|T]$, whereas in $lastElem(A, T, [B|V])$, parameter $[B|V]$ is *not* constructed using parameters 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 executions shown here, the schema underlying Step 1 is a divide-and-conquer schema, a quite general expression of which is as follows:

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

Parameter X of r is the induction parameter (in the sense that it is decomposed for recursive calls), parameter Y is the optional "result" parameter (in the sense that it is constructed from partial results TY_i obtained through recursion), and parameter Z is the optional "passive" parameter (in the sense that it is not decomposed for recursive calls, but serves to solve the base case and/or to combine 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 [3]. Step 1 of the basic synthesis algorithm instantiates the h, t, r, *solve*, and *decompose* "place-holders," so that the undefined relation actually is the *compose* place-holder.

Let us continue now and show an unsuccessful execution 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, now):

$delOddElems([\,], [\,]) \leftarrow$
$delOddElems([1], [\,]) \leftarrow$ $\qquad\qquad \leftarrow delOddElems([5], [5])$
$delOddElems([2], [2]) \leftarrow$
$delOddElems([3, 4], [4]) \leftarrow$
$delOddElems([6, 7, 8], [6, 8]) \leftarrow$

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

$delOddElems([\,], [\,]) \leftarrow$
$delOddElems([HL|TL], R) \leftarrow delOddElems(TL, TR), combine(HL, TR, R)$

Suppose Step 2 somehow abduces the following evidence for the undefined relation:

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

Step 3 induces $combine(X, T, V) \leftarrow$ as lgθ of the positive evidence, which is not "acceptable." However, recursive invocation of the basic synthesis algorithm on 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:

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

In other words, *combine* is defined as the conjunction of *several* clauses, with bodies involving relations other than *combine*, 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 (which 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 only using the equality relation.

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, by showing how multi-clausal (i.e. conjunctive) definitions 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 a *set* of clauses c_i, such that each c_i is the 'classical' lgθ of a subset $\mathcal{C}_i$ of $\mathcal{C}$ and such that the union of the $\mathcal{C}_i$ is $\mathcal{C}$. The clauses in $\mathcal{C}_i$ ought to be two-by-two "compatible," in the sense that they construct their "result parameters" in the same way. "Compatibility" is achieved if the 'classical' lgθ of $\mathcal{C}_i$ 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." 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: this is an undecidable property [19] 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 Steps 1 or 2 provide 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 discriminating literals with relations other than equality. This means that 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 Sections 4 and 5, we review related work and outline future work, respectively, and finally we conclude in Section 6.

2 Basic Concepts

After introducing the used notation 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).

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, $\langle f(X,Y), g(X,Y,Z)\rangle$ is a term representing the couple (or: 2-tuple) built of the two terms $f(X,Y)$ and $g(X,Y,Z)$.

When we do not want to (or cannot) fix the arity of a relation symbol, we use a "..." notation in conjunction with subscripted variables (subscripts starting 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 this paper, a very simple generalization model will suffice, namely θ-subsumption [18]. Let us define it step by step, in a first-order logic setting.

Definition 1 (Term/literal generality)
A term/literal g is *more general than* term/literal s iff there exists a substitution σ such that $s = g\sigma$. We also say that g is *less specific than* s, that s is *less general than* g, or that s is *more specific than* g. For literals, the two considered literals must be of the same sign, the same relation symbol, and the same arity.

For instance, term/literal $f(X,4,Y)$ is more general than term/literal $f(2,4,Z)$, with $\sigma = \{X/2, Y/Z\}$.

This generality relation induces partial orders on the term and literal sets, the former having all variables as greatest elements, the latter having no greatest element. Let us give names to the least-upper-bound operators for these partially ordered sets, and recall their generic definitions. The *least general generalization*, or simply *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. Similarly for the least generalization of two literals a and b, denoted by $lgl(a,b)$, except that they must have the same sign, the same relation symbol, and the same arity; otherwise, their least generalization is undefined. A more constructive definition of these operators emerges as a property.

Property 1 (Least generalization, under θ-subsumption, of two terms/literals)
For terms:

$$lgt(f(s_1,\ldots,s_n), g(t_1,\ldots,t_m)) = \begin{cases} f(lgt(s_1,t_1),\ldots,lgt(s_n,t_n)), & \text{if } f/n = g/m \\ V, & \text{otherwise,} \end{cases}$$

where V is a new variable that will represent the two terms throughout the context.
Similarly for positive literals (or: atoms): [2]

$$lgl(p(s_1,\ldots,s_n), q(t_1,\ldots,t_m)) = \begin{cases} p(lgt(s_1,t_1),\ldots,lgt(s_n,t_n)), & \text{if } p/n = q/m \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Similarly for negative literals:

$$lgl(\neg p(s_1,\ldots,s_n), \neg q(t_1,\ldots,t_m)) = \begin{cases} \neg p(lgt(s_1,t_1),\ldots,lgt(s_n,t_n)), & \text{if } p/n = q/m \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

[2] Note the use of the *lgt* operator on the right-hand side!

Computing the least generalization of two terms/literals is also known as *anti-unification*, and its result as their *least anti-instance*. This is so because computing the greatest-lower-bound of two terms/literals is nothing else but unification, which yields their most general specialization (or: greatest instance) via the application of a substitution called the most-general unifier. The least generalization of two terms/literals is unique (up to variable renaming), if it is defined. For instance, 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): this is so because otherwise the two given terms/atoms would not be more specific than their least generalization.

We can now define a simple model of generality for clauses [18]. We assume that clauses are seen as sets of (positive and negative) literals.

Definition 2 (Clause θ-subsumption)
A clause g *θ-subsumes* a clause s iff there exists a substitution σ such that $g\sigma \subseteq s$. We also say that g is *more general* than s under θ-subsumption, that g is *less specific* than s under θ-subsumption, that s is *less general* than g under θ-subsumption, or that s is *more specific* than g under θ-subsumption. Two clauses are *θ-subsumption-equivalent* iff they θ-subsume each other. A clause is *reduced* iff it is θ-subsumption-equivalent to no proper subset of itself.

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 $\sigma = \{I/X, L/[HL|TL]\}$; but the converse is not true.

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. This may happen when g and s are recursive. For instance, take g as $p(f(X)) \leftarrow p(X)$ and s as $p(f(f(X))) \leftarrow p(X)$. This is why θ-subsumption is only an approximation of a generality model, but a correct one, and even a sufficient one for our purposes (as we do not consider recursive clauses) [18].

Every set of θ-subsumption-equivalent clauses has a unique (up to variable renaming) reduced representative [18].

As a partial order, the θ-subsumption relation induces a lattice on the clause set, with the empty clause as unique top element. A constructive definition of the least upper-bound-operator for this lattice emerges as a property [18].

Property 2 (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 $combine(I, L, [I|L]) \leftarrow even(I)$. In general, the resulting clause is not necessarily reduced, so a reduction algorithm may then need to be run to produce a reduced θ-subsumption-equivalent clause [18].

So far, we have only characterized the least generalization (under θ-subsumption) of *two* clauses, but we will also need to compute least generalizations of non-empty *sets* of clauses. Again, a constructive definition of this operator emerges as a property.

Property 3 (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.[3]

This is the 'classical' definition of this concept. In Section 3.2, we will propose a redefinition thereof.

A useful property linking the previous two operators arises.

Property 4 If $c \in \mathcal{C}$, then $lg\theta(lg\theta(\mathcal{C}), c) = lg\theta(\mathcal{C})$.

In this paper, we will not need a concept of relative θ-subsumption (as we only want generalizations in the absence of background knowledge). Also, we will only consider definite clauses, rather than full clauses.

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

2.3 Construction Modes

Informally, 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 instance, in *append*, the third parameter is mandatorily constructed from the first two parameters. In *combine* (see Example 3 above), the third parameter is mandatorily constructed from the second parameter and optionally from the first. We now incrementally define the notion of construction mode.

2.3.1 Syntactic Construction

Let us first define some notions about syntactic construction.

Definition 3 (Leaves and vertices of a term)
The *leaves* of a term t, denoted by $leaves(t)$, are the set of the variables and constants occurring in t.
The *vertices* of a term t, denoted by $vertices(t)$, are the multi-set of the variables and function symbols (including the constant symbols) occurring in t.

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

Definition 4 (Syntactic construction)
Term s is *syntactically obtained from* term t iff $leaves(t) \subseteq leaves(s)$. We denote this by $t \subseteq s$.
Term s *syntactically contains* term t iff $vertices(t) \sqsubseteq vertices(s)$, where $\sqsubseteq$ denotes multi-set inclusion. We denote this by $t \sqsubseteq s$.

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

For atoms of a given relation, one can express syntactic construction constraints: this will be the role of construction modes (defined below).

The reason why we sometimes consider function symbols of arity higher than 0 (rather than just constants) is that we want to achieve that $f(a, b) \not\sqsubseteq [a, b]$. Similarly, the reason why we sometimes consider multi-sets is that we want to achieve that $[a, b, b] \not\sqsubseteq [a, b]$. Finally, note that the two concepts are much more general than the sub-term (i.e. sub-tree) concept, and this additional generality is crucial in many cases. For instance, in atom $efface(d, [f, e, d], [f, e])$, the vertices of $[f, e]$ are a sub-multi-set of the vertices of $[f, e, d]$, but $[f, e]$ is not a sub-tree of $[f, e, d]$.

2.3.2 Semantic Construction

In order to capture more than just syntactic construction (which is basically achieved using equality ($=$) only, inside a single atom), we have to extend this notion to *definite clauses*. Indeed, body atoms may perform some computations of *semantic* construction of the head parameters, using relations other than equality. These atoms cannot be "forward-compiled" into the head of the clause, unlike equality atoms. For instance, in $min(X, Y, X) \leftarrow X \leq Y$, one cannot "forward-compile" $X \leq Y$ into $min(X, Y, X)$. Also, parameter Y does not syntactically contribute to constructing result X (the third parameter), but it does so semantically.

2.3.3 Construction Modes

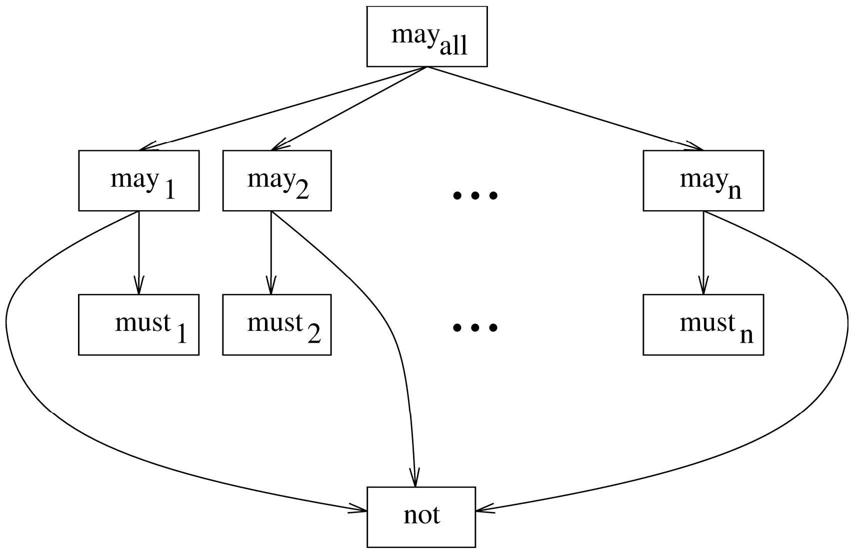
Definition 5 (Construction modes)
A *construction mode* m for a relation r of arity n is a total function from the set $\{1, 2, \ldots, n\}$ into the set $\{may_1, \ldots, may_n, may_{all}, must_1, \ldots, must_n, res_1, \ldots, res_n, not\}$, such that res_j is in the range of m iff may_j or $must_j$ also is in the range of m, and such that every res_j is at most once in the range of m. We also say that $m(i)$ is the *mode* of the i^{th} parameter of r.

A construction mode m is often written in the more suggestive form $r(m(1), \ldots, m(n))$. Do not confuse the position i of a parameter and the index j of its mode $m(i)$, say $must_j$. Since we do not consider input/output modes in this paper, we often simply speak about modes here. For instance, $combine(may_1, must_1, res_1)$ is a mode for *combine*.

In a first approximation, the intended semantics of a mode is as follows:

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

We will refine (for semantic construction) and formalize all this hereafter, via the concept of admissibility. Note that mode may_{all} sort-of "generalizes" every may_j, which itself "generalizes" $must_j$ and *not*, and is thus always "safe" to use. The following figure illustrates this:



For instance, in $sister(P, Q)$, neither parameter is syntactically constructed from the other one, so its mode could be $sister(may_{all}, may_{all})$ in addition to the more "specific" $sister(not, not)$. However, the more "specific" a mode, the more useful it may be. Note that a mode is thus not necessarily unique for a given relation.

Let us now see a few other examples of construction modes. We drop the indexes j of may_j, $must_j$, and res_j when they all have the same value.

Example 4 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 TY_i being obtained through recursion, they must all somehow be used to construct Y, because some of the recursive calls would otherwise have been useless.

The HX_i need not always be used to construct Y, as it depends on the particular program. For instance, the mode for $combine(HL, TR, R)$ (see Example 3) is $combine(must, must, res)$, whereas 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 $add1(not, must, res)$. 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 $cons'(must, must, res, not)$. 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 $cons''(not, must, res, must)$. So there also is no fixed mode for the passive parameter, and its most general mode thus 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

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

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

$$r(may, may, may).$$

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 version 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), and are beyond the scope of this paper, our objective here being merely to establish some simple concepts. $\square$

The key issue is that modes can easily be pre-computed for 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.

The definition of construction modes itself can be further generalized, allowing for instance a set of modes for every argument position, rather than just a single mode. This would allow the expression of a mode like $intersection(\{must_1, must_2\}, \{res_1\}, \{res_2\})$ for $intersection(I, A, B)$ (which holds iff set I is the intersection of sets A and B). We do not consider such modes in this introductory paper, but the corresponding generalization is straightforward.

2.4 Admissibility

In a first version, the concept of admissibility captures the notion of what it means for an *atom* to satisfy a construction mode for its relation. After refining a definition for this concept, based purely on syntactic construction, we will generalize this concept and define what it means for a *definite clause* to satisfy a construction mode for the relation in its head, and add considerations of semantic construction.

2.4.1 Syntactic Admissibility of an Atom wrt a Construction Mode

Let m be a mode for a relation r, and let $r(t_1, \ldots, t_n)$ be the considered atom, where n is a natural number. Let the indexes in m run from 1 to k inclusive, where k is a natural number. Let

$Must_j = \langle t_i \mid m(i) = must_j \rangle$, and let $Must = \langle t_i \mid m(i) = must_j$ for some $j\rangle$. Similarly for May_j, May_{all}, May, Res_j, Res, and Not.

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

According to the given informal approximate semantics of modes, for admissibility of atom $r(t_1, \ldots, t_n)$ wrt mode m, we first need to express that every parameter in the corresponding position of $must_j$ is mandatory in syntactically constructing the parameter in the corresponding position of res_j. Here, we prefer to use syntactic containment as actual instance of syntactic construction.[4] Formally:

$$\forall 1 \leq j \leq k : Must_j \sqsubseteq Res_j \tag{1}$$

For instance, this is the case for the *compose* atom and mode mentioned above. Note that k may be 0, such as in $sister(may_{all}, may_{all})$. Condition (1) then trivially holds.

Next, we need to express that every parameter in the corresponding position of may_j is optional for syntactically constructing the parameter in the corresponding position of res_j, and that every parameter in the corresponding position of may_{all} is optional for syntactically constructing any of the parameters in the corresponding positions of all res_j. By themselves, these requirements lead to no formula, because of the optional nature of this syntactic construction. But if we refine the given approximate semantics by also requiring that the parameter in the corresponding position of res_j can *only* be syntactically obtained from[5] the parameters in the corresponding positions of may_j, may_{all}, and $must_j$, then we can formalize this as follows:

$$\forall 1 \leq j \leq k : Res_j \subseteq \langle May_j, May_{all}, Must_j \rangle \tag{2$'$}$$

So no leaves may be "invented" when building each Res_j.

For instance, the atom $addOne(a, 0, s(0))$ satisfies condition (2′) for the construction mode $addOne(may_1, must_1, res_1)$ because: $\langle s(0) \rangle \subseteq \langle \langle a \rangle, \langle \rangle, \langle 0 \rangle \rangle$.

However, this requirement is a bit too strong, as new leaves *do* sometimes appear in parameters with mode res_j. For instance, a base constant of the type of such a parameter may appear: the atom $addPlateau(a, [\,], [a, s(0)])$ does not satisfy condition (2′) for mode $addPlateau(may_1, must_1, res_1)$, because base constant 0 is "invented" by the parameter with mode res_1. Since such base constants cannot really be considered new, we should add them to the right-hand side:

$$\forall 1 \leq j \leq k : Res_j \subseteq \langle May_j, May_{all}, Must_j, 0, nil, \ldots \rangle \tag{2$''$}$$

Also, two parameters with modes res_j and res_g respectively (where $j \neq g$) may share a new leaf: the atom $compose_2([b, c], [b], [a, b, c], [a, b])$ does not satisfy condition (2″) for the construction mode $compose_2(must_1, must_2, res_1, res_2)$, because both res_i parameters "invent" the same constant, namely a. So shared leaves should also be exempted from the requirement above. This can only be achieved by eliminating the "iteration" $1 \leq i \leq k$ from (2″): [6]

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

We do not allow for non-base non-shared constant leaves to be invented by res_i parameters, mostly because we have not yet encountered such a relation (such that it has a recursive definition).

Last, we would theoretically need to express that every parameter in the corresponding position of *not* is not used at all in syntactically constructing any of the parameters in the corresponding positions of all res_j. However, to reduce the computations of admissibility checking, we decided not to formulate such a negative check, thus essentially giving the *not* mode a "do not care" semantics.

To summarize so far: an atom $r(t_1, \ldots, t_n)$ is admissible wrt a mode m for r iff conditions (1) and (2) above are satisfied.

[4] This choice is largely motivated by the role of parameters with mode $must_j$ in the *compose* place-holder of the divide-and-conquer schema. Indeed, these are the partial results obtained by recursive calls, so these recursive calls would somehow be wasted if these values were not entirely used (i.e., using *all* their variables and functors) in constructing the corresponding res_j parameter. But we conjecture that this is a very natural and general choice.

[5] Again, this choice is largely motivated by the *compose* place-holder of the divide-and-conquer schema.

[6] Let $sharedLeaves(t)$ designate the set of leaves shared by all components of tuple t.

2.4.2 Semantic Admissibility of a Definite Clause wrt a Construction Mode

Let $r(t_1, \ldots, t_n) \leftarrow \mathcal{B}$ be a definite clause, where $\mathcal{B}$ represents 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 $insert(X, [Y|L], [X, Y|L]) \leftarrow X \leq Y$. We also constrain the clause to be non-recursive (this will be motivated in Section 5).

Definition 6 (Proper and reconcilable clauses)
We refer to an equality-free non-recursive definite clause as a *proper clause.* Two proper clauses are *reconcilable* iff they define the same relation (i.e. have the same relation in their heads).

Basically, the reasoning is the same as for atoms. In the head of the clause, every parameter with mode $must_j$ is mandatory in constructing the parameter in the corresponding position of res_j, whether the construction is syntactic or semantic. So equation (1) is adapted as follows:

$$\forall 1 \leq j \leq k : Must_j \sqsubseteq \langle Res_j, \mathcal{B}' \rangle \tag{3}$$

where $\mathcal{B}'$ is a tuple built of the atoms (seen as terms) of conjunction $\mathcal{B}$. Similarly, (2) becomes:

$$leaves(Res) \setminus sharedLeaves(Res) \subseteq leaves(\langle May, May_{all}, Must, \mathcal{B}' \rangle) \cup \{0, nil, \ldots\} \tag{4}$$

For instance, the clause $min(X, Y, X) \leftarrow X \leq Y$ satisfies conditions (3) and (4) for the construction mode $min(may_1, must_1, res_1)$, but not (the old) condition (1), because Y does not syntactically contribute to constructing result X, though it does so semantically, as testified by the fact that (3) holds.

Now we can finally propose the following definition of admissibility.

Definition 7 (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 iff conditions (3) and (4) above are satisfied.
A set of reconcilable clauses is *admissible* wrt a mode m for the relation in their heads iff each of its clauses is admissible wrt m.

Lemma 1 (Preservation of admissibility under θ-subsumption)
If proper clause c is admissible wrt some mode m for the relation in its head, and if c θ-subsumes proper clause d, then d is also admissible wrt m.

Proof: Let σ be the witness substitution under which c θ-subsumes d (see Definition 2): $c\sigma \subseteq d$ (remember that clauses are here seen as atom sets). Supposing c has the structure $r(t) \leftarrow \mathcal{B}$, for some tuple t and body $\mathcal{B}$, this all means that d has the structure $r(t)\sigma \leftarrow \mathcal{B}\sigma, \mathcal{D}$, for some atom conjunction $\mathcal{D}$. Since c is admissible wrt mode m for r, i.e. since conditions (3) and (4) are satisfied for $r(t) \leftarrow \mathcal{B}$, the conditions (3)$\sigma$ and (4)σ are also satisfied for $r(t)\sigma \leftarrow \mathcal{B}\sigma$, by the rule of universal instantiation, i.e. this proper clause (which is $c\sigma$) is also admissible wrt m. Since d is known to be a proper clause and since its only difference with $c\sigma$ is $\mathcal{D}$, the sets in the right-hand sides of conditions (3) and (4) can only become larger, whereas their left-hand side sets are unchanged; so the truth of these conditions is maintained for d. So we can conclude that d is also admissible wrt m. $\square$

We now prove a theorem establishing a sufficient criterion for deciding whether a clause set is admissible or not.

Theorem 1 (Sufficient criterion for clause set admissibility)
Let $\mathcal{C}$ be a non-empty set of reconcilable clauses, and let m be a mode for the relation in their heads. If $lg\theta(\mathcal{C})$ is admissible wrt m, then $\mathcal{C}$ is admissible wrt m.

Proof: Let $lg\theta(\mathcal{C})$ be admissible wrt m. Since $\mathcal{C}$ is made of reconcilable clauses for r, it follows from Property 2 that $lg\theta(\mathcal{C})$ itself is a definite clause for r. Also, by definition, $lg\theta(\mathcal{C})$ θ-subsumes all clauses in $\mathcal{C}$. So let d be an arbitrary clause in $\mathcal{C}$; we have that $lg\theta(\mathcal{C})$ θ-subsumes d. By Lemma 1, d is admissible wrt m. Since d was chosen arbitrarily, we can conclude that *all* clauses of $\mathcal{C}$ are admissible wrt m, i.e. that $\mathcal{C}$ is admissible wrt m. □

Note that the converse of this theorem is not true. For instance, the set $\{insert(1,[2],[1,2]) \leftarrow\ , insert(4,[3],[3,4]) \leftarrow\}$ is admissible wrt the construction mode $insert(may, must, res)$, but its least generalization $insert(X,[Y],[K,M]) \leftarrow$ is not admissible wrt that mode.

2.5 Compatibility

Given a set of reconcilable clauses with relation r in their heads, and given a mode m for r, we now want to define a relationship over this set, such that two clauses are related iff they construct their parameters with mode res_j in a "similar" way. This can be done via the concept of compatibility.

Definition 8 (Clause compatibility)
Two reconcilable clauses c and d are *compatible* (*with each other*) wrt a mode m for the relation in their heads iff $lg\theta(c,d)$ is admissible wrt m. We also say that c is *compatible with* d, and vice-versa.

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

Note that compatibility is thus a problem-independent concept: it is parameterized on the problem-dependent definitions of admissibility and construction modes.

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 not reflexive, because its definition does not require the two given clauses to be admissible wrt the given mode. For instance, the clause $q(d,[f],[e,f])$ is not admissible wrt the mode $q(may, must, res)$, therefore it is not compatible with itself wrt that mode. The reason why the two given clauses are not required to be admissible wrt the given mode is that we thus do not have to verify or ensure this before checking compatibility. We can do so because reflexivity is an unnecessary property for our purposes on compatibility.

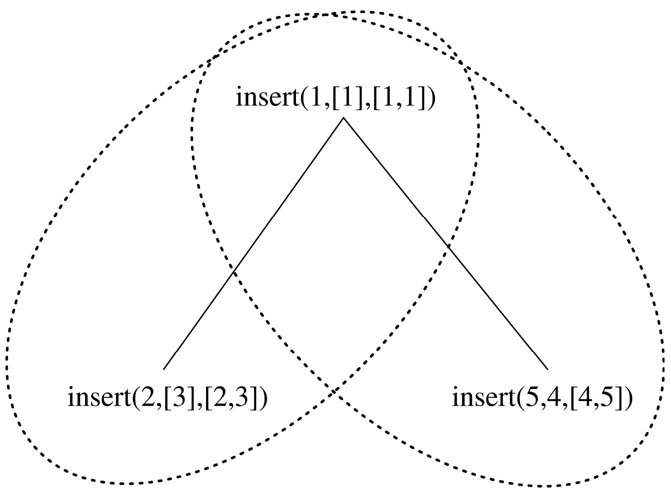
The compatibility relation is symmetric by construction, because the definition is based on least generalizations (under θ-subsumption), which already have this property.

Finally, the compatibility relation is not transitive. For instance, the clauses $insert(1,[1],[1,1]) \leftarrow$ and $insert(2,[3],[2,3]) \leftarrow$ are compatible wrt $insert(must, must, res)$, because their lgθ, namely $insert(X,[Y],[X,Y]) \leftarrow$, is admissible; also, the clauses $insert(1,[1],[1,1]) \leftarrow$ and $insert(5,[4],[4,5]) \leftarrow$ are compatible, because their lgθ, namely $insert(X,[Y],[Y,X]) \leftarrow$, is admissible; but $insert(2,[3],[2,3]) \leftarrow$ and $insert(5,[4],[4,5]) \leftarrow$ are not compatible, because their lgθ, namely $insert(X,[Y],[Z,K]) \leftarrow$, is not admissible. Upon close inspection of the desired concept of compatibility, it turns out that transitivity is impossible to achieve without sacrificing most of the power of the *Program Closing Method* described later. Indeed, if compatibility were transitive, then some clauses would be found to be compatible (e.g. the last two clauses above), although they do not construct their res_j parameters in a similar way.

So far, we have only defined the compatibility of *two* clauses, but we will also need a notion of compatibility of a non-empty *set* of clauses.

Definition 9 (Clause set compatibility)
A non-empty set $\mathcal{C}$ of reconcilable clauses is *compatible* wrt a construction mode m for the relation in their heads iff any two clauses in $\mathcal{C}$ are compatible wrt m.

Note that a singleton set is thus trivially compatible wrt *any* mode. Also notice that a compatible subset of a set $\mathcal{D}$ of reconcilable clauses is a *clique* (or: *maximal connected component*) of the graph with node set $\mathcal{D}$ and edge set induced by the compatibility relationship. For instance, the set $\mathcal{D} = \{insert(1,[1],[1,1]) \leftarrow, insert(2,[3],[2,3]) \leftarrow, insert(5,4,[4,5]) \leftarrow\}$ has the following compatibility graph:



where the edges are induced by the compatibility relationship on this set, and the contours identify the cliques.

Note that, due to the desirable absence of (reflexivity and) transitivity of compatibility, cliques are not simply equivalence classes of an equivalence relation: this considerably complicates clique finding, as seen in Section 3.

Checking whether a given set is compatible seems to be an extremely tedious and time-consuming operation. Fortunately, the following theorem shows how to shortcut such a decision process (for sets of at least two clauses, because, as pointed out, a singleton set is trivially compatible wrt any mode, so no decision procedure is needed then).

Theorem 2 (Sufficient criterion for clause set compatibility)
Let $\mathcal{C}$ be a set of at least two reconcilable clauses, and let m be a mode for the relation in their heads. If $lg\theta(\mathcal{C})$ is admissible wrt m, then $\mathcal{C}$ is compatible wrt m.

Proof: Let $lg\theta(\mathcal{C})$ be admissible wrt m. Let c and d be two arbitrary clauses in $\mathcal{C}$. By construction, $lg\theta(\mathcal{C})$ θ-subsumes $lg\theta(c, d)$. So, by Lemma 1, $lg\theta(c, d)$ is admissible wrt m. That is, by Definition 8 and since c and d are reconcilable, clauses c and d are compatible wrt m. Since c and d were chosen arbitrarily, set $\mathcal{C}$ is compatible wrt m, by Definition 9. □

Also, the following theorem shows that the least generalization (under θ-subsumption) of a clause set is compatible with every element of that set. (This is trivial for singleton sets, so we restrict the theorem to sets of at least two elements, so that the proof can use the previous theorem.)

Theorem 3 (Representative of a compatible clause set)
Let $\mathcal{C}$ be a set of at least two reconcilable clauses, and let m be a mode for the relation in their heads. We have that $\mathcal{C}$ is compatible wrt m iff $\mathcal{C} \cup \{lg\theta(\mathcal{C})\}$ is compatible wrt m.

Proof: First assume that $\mathcal{C}$ is compatible wrt m. Then we have that:
$\mathcal{C} \cup \{lg\theta(\mathcal{C})\}$ is compatible wrt m
iff any clause $c \in \mathcal{C}$ is compatible with $lg\theta(\mathcal{C})$ wrt m (by Definition 9 and by the assumption)
iff $lg\theta(lg\theta(\mathcal{C}), c)$ is admissible wrt m (by Definition 8)
iff $lg\theta(\mathcal{C})$ is admissible wrt m (by $c \in \mathcal{C}$ and Property 4)
implies $\mathcal{C}$ is compatible wrt m (by Theorem 2)
iff true (by the assumption).
Conversely, assume that $\mathcal{C} \cup \{lg\theta(\mathcal{C})\}$ is compatible wrt m. Then we immediately have that $\mathcal{C}$ is compatible wrt m (as any subset of a set compatible wrt m is compatible wrt m). □

This basically means that the lgθ of a compatible set can be taken as a representative of that set, because it constructs its parameters with mode $must_j$ in a way "similar" to how this is done for *each* element in that set. This is the crucial idea behind our *Program Closing Method*, which is presented next.

3 The Program Closing Method

Given:

- an open logic program $\mathcal{Q}$ for a relation q, in the sense that it has no clauses for one used relation, say r,
- a set $\mathcal{E}$ of reconcilable clauses defining r, called the *evidence set*,
- a construction mode m for r,

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

This problem statement is not quite the same as the one of the general ILP task. Of course, we do not want $\mathcal{R}$ to be equal to $\mathcal{E}$, nor to cover all syntactically possible atoms for r. What is wanted is rather that $\mathcal{R}$ covers an "extension" of $\mathcal{E}$, such that this "extension" coincides with the unknown intended relation r.

In Section 3.1, we present the *Program Closing Method* itself, and in Section 3.2, we introduce an acceptability criterion for the program $\mathcal{R}$ inferred by that method.

3.1 The Method

Basically, the idea of the *Program Closing Method* comes from Theorem 3: one can divide $\mathcal{E}$ into a minimal number of subsets that are compatible wrt m, and then take the set of their least generalizations (under θ-subsumption) as representatives, i.e. as $\mathcal{R}$. Note that $\mathcal{R}$ may thus be conjunctively defined, as desired.

Let us first get a feeling for the desired algorithm, via some examples.

Example 5 For *combine* of Example 3, let $\mathcal{E}^+$ be the following positive evidence set:

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

Also let $\mathcal{E}^-$ be the set containing the negative evidence, namely $\{\leftarrow combine(5,[\,],[5])\}$, and let the mode m be $combine(may, must, res)$ (as dictated in general for the *compose* place-holder of the divide-and-conquer schema, see Example 4).

If we check the compatibility of the *entire* set $\mathcal{E}^+$ wrt m, using Theorem 2, we obtain that $lg\theta(\mathcal{E}^+)$ is $combine(X,T,V) \leftarrow$, which is not admissible wrt m. So we cannot judge, using Theorem 2, whether $\mathcal{E}^+$ is compatible or not. But $\mathcal{E}^+$ is *not* compatible, because E_1 and E_2 are not compatible. So we need a division of $\mathcal{E}^+$ into a minimal number of subsets such that each one is compatible wrt m. In this case, the 2 subsets $\mathcal{E}_1 = \{E_1, E_3\}$ and $\mathcal{E}_2 = \{E_2, E_4\}$ are compatible wrt m, because both $lg\theta(\mathcal{E}_1)$, which is $combine(X,T,T) \leftarrow$, and $lg\theta(\mathcal{E}_2)$, which is $combine(X,T,[X|T]) \leftarrow$, are admissible wrt m. Divisions into 3 or 4 compatible subsets are also possible, but they would not be "minimal." Therefore, we get the following program for *combine*:

$$
\begin{array}{l}
combine(X,T,T) \leftarrow \\
combine(X,T,[X|T]) \leftarrow
\end{array}
$$

by collecting the $lg\theta$s of the 2 identified subsets. However, this program is not acceptable because the atom of the negative evidence is covered by the second clause. To avoid this, there are two solutions. First, there could be other open relations in the *delOddElems* program, and they would discriminate between these two clauses for *combine*. The aim of our *Program Closing Method* is not to infer the discriminants *odd* and *even* from the given evidence, so another method would need to be invoked to do so (e.g., see the *Proofs-as-Programs Method* of the SYNAPSE technique [6, 3], also see Section 4.1). Second, the present method could be given more informative evidence, such as the following new set $\mathcal{E}'^{+}$:

$combine(1, [\,], [\,]) \leftarrow odd(1)$	(E_1)
$combine(2, [\,], [2]) \leftarrow even(2)$	(E_2)
$combine(3, [4], [4]) \leftarrow odd(3)$	(E_3)
$combine(6, [8], [6, 8]) \leftarrow even(6)$	(E_4)

Let $\mathcal{E}'^{-}$ also be $\{\leftarrow combine(5, [\,], [5])\}$. If we check the compatibility of the entire set $\mathcal{E}'^{+}$ wrt m, we obtain that $lg\theta(\mathcal{E}'^{+})$ is $combine(X, T, V) \leftarrow$, which is not admissible wrt m. So we need a division of $\mathcal{E}'^{+}$ into a minimal number of subsets such that each one is compatible wrt m. In this case, the subsets $\mathcal{E}'_1 = \{E_1, E_3\}$ and $\mathcal{E}'_2 = \{E_2, E_4\}$ are compatible wrt m, because both $lg\theta(\mathcal{E}'_1)$, which is $combine(X, T, T) \leftarrow odd(X)$, and $lg\theta(\mathcal{E}'_2)$, which is $combine(X, T, [X|T]) \leftarrow even(X)$, are admissible wrt m. Therefore, we get the following program for *combine*:

$combine(X, T, T) \leftarrow odd(X)$
$combine(X, T, [X|T]) \leftarrow even(X)$

which is acceptable now, because the atom of the negative evidence is not covered by this program, assuming that *odd* and *even* are primitives. □

Example 6 Let $\mathcal{E}^{+}$ be the following evidence set for *insert*:

$insert(3, [3], [3, 3]) \leftarrow$	(E_1)
$insert(1, [2], [1, 2]) \leftarrow$	(E_2)
$insert(2, [1], [1, 2]) \leftarrow$	(E_3)

and let the mode m be $insert(must, must, res)$. If we check the compatibility of the entire set $\mathcal{E}^{+}$ wrt m, we obtain that $lg\theta(\mathcal{E}^{+})$ is $insert(X, [Y], [Z, T]) \leftarrow$, which is not admissible wrt m. So we need a division of $\mathcal{E}^{+}$ into a minimal number of subsets such that each one is compatible wrt m. In this case, the subsets $\mathcal{E}_1 = \{E_1, E_2\}$ and $\mathcal{E}_2 = \{E_1, E_3\}$ are compatible wrt m, because both $lg\theta(\mathcal{E}_1)$, which is $insert(X, [Y], [X, Y]) \leftarrow$, and $lg\theta(\mathcal{E}_2)$, which is $insert(X, [Y], [Y, X]) \leftarrow$, are admissible wrt m. Note that this division is *not* a partition. Therefore, we get the following program for *insert*:

$insert(X, [Y], [X, Y]) \leftarrow$
$insert(X, [Y], [Y, X]) \leftarrow$

which is acceptable, but of course only an approximation, because the initial evidence is not very informative. □

In these two examples, while we are trying to find some subsets of the given positive evidence set such that they are compatible wrt the given mode, we are actually trying to find the minimal number of cliques of the compatibility graph over this set such that these cliques contain all the given positive evidence. Note that $\mathcal{E}_1$ and $\mathcal{E}_2$ do not form a partition in Example 6, while they do form a partition in Example 5. Actually, in Example 6, we could have used a partition such as $\mathcal{E}_1 = \{E_2\}$ and $\mathcal{E}_2 = \{E_1, E_3\}$, but we preferred a cover such as $\mathcal{E}_1 = \{E_1, E_2\}$ and $\mathcal{E}_2 = \{E_1, E_3\}$, because the more evidence in a subset, the "better" its least generalization is (that is, the more further evidence it covers). Therefore, what we try to find is a *node clique cover* (NCC) of the compatibility graph of the given evidence set for a given relation wrt a given mode. Finding a node clique cover of a graph is referred to as the *node clique cover problem* (or shortly *clique cover problem*) in graph theory: we try to find the minimum number of cliques that cover the nodes of the given undirected graph, such that a node already covered by a clique may also be covered by another clique [12]. This is an NP-complete problem. Note that we do not try to find a minimum set cover, i.e. the smallest subset $\mathcal{S}$ of the given subsets of a finite set $\mathcal{U}$ such that the union of the members of $\mathcal{S}$ is equal to $\mathcal{U}$, which is a more general problem. Indeed, finding an NCC fits our goal better, in the sense that it is more natural to represent the compatibility relation wrt the given mode amongst the evidence set by a graph (as edges) rather than by a set; and this makes finding an NCC more efficient than finding a minimal set cover.

Now we can finally introduce our re-definition of the concept of least generalization (under θ-subsumption) of a set of clauses:

Definition 10 (Least generalizations,[7] under θ-subsumption, of a clause set)
The *least generalizations under θ-subsumption* of a non-empty set $\mathcal{C}$ of clauses wrt an over-generality criterion G, denoted by $lgs\theta(\mathcal{C}, G)$, are 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 G.

It remains to see how to (efficiently) solve the NCC problem. The NCC problem being an NP-complete problem, there does not exist a polynomial time algorithm for it, assuming $P \neq NP$. We approach NP-complete problems in two ways [8]:

- try to find as much improvement as possible over straightforward exhaustive search to find the optimal solution, such as by using branch-and-bound or dynamic programming;
- try to find a "good" solution within an acceptable amount of time, using problem-specific methods, called "heuristics," such as neighborhood search, in which, using a pre-selected set of local operators, an initial solution is improved repeatedly until a "local optimum" solution, i.e. a "good" solution, has been obtained.

We used the first approach in [2], but it is very inefficient in time for our purposes, so we now choose the second approach, that is, we use an approximation algorithm for the NCC problem.

The literature has a number of heuristic algorithms for the graph coloring problem, from which we can construct algorithms for the NCC problem. In [12], an algorithm for *edge clique covering* (ECC) is constructed without using the heuristics for graph coloring, but rather a heuristic introduced by Kellerman for the keyword conflict problem. We used the approximation algorithm introduced in [12] for the ECC problem to construct an approximation algorithm for the NCC problem. Note that there is an approximation algorithm for the NCC problem iff there is an approximation algorithm for the ECC problem [12].

We now switch the terminology to our particular problem, and talk about clauses (of the evidence) instead of nodes, and about compatibility (wrt a mode) instead of connectedness via an edge. The approximation algorithm forms the minimal number of cliques, labeled $C_1, \ldots, C_p$, examining the clauses one by one. In the description of the algorithm, *CurrentClause* is the label of the current clause being examined, and k is the number of the cliques that have been created so far. The heuristic is that when a clause labeled i is being examined, the next clauses to be examined are the ones that are compatible with i wrt mode m, and labeled j, where $j < i$. In the description of the algorithm, *ClausesToBeExamined* is the set of the labels of the next clauses to be examined when the clause labeled *CurrentClause* is being examined.

```
Algorithm ClauseCliqueCover
Inputs:
– a set $\mathcal{E}$ of $n$ reconcilable clauses (defining a relation $r$), whose elements $E_i$ are labeled from 1 to $n$;
– a construction mode $m$ for $r$.
Output:
– a set $\mathcal{C}$ of cliques covering $\mathcal{E}$, labeled $C_1, \ldots, C_p$, with $p$ minimum such that each $C_i$ is compatible wrt $m$.
 Initialize the number of cliques: $k \leftarrow 0$;
 Initialize the label of the current clause being examined: $CurrentClause \leftarrow 1$;
 while $CurrentClause \leq n$ do
   begin
     $ClausesToBeExamined \leftarrow \{j | j < CurrentClause \wedge compatible(CurrentClause, j, m)\}$;
     if $ClausesToBeExamined = \{\}$ then
       begin {Create a new clique}
         $k \leftarrow k + 1$;
         $C_k \leftarrow \{CurrentClause\}$;
         $CurrentClause \leftarrow CurrentClause + 1$
```

[7] Note the plural!

```
      end ;
    else {Try to insert $CurrentClause$ into existing cliques $C_1, \ldots, C_k$}
      begin
        Initialize the clique into which $CurrentClause$ may be inserted: $l \leftarrow 1$;
        Initialize the union of cliques into which $CurrentClause$ is inserted: $\mathcal{V} \leftarrow \{\}$;
        while $l \leq k$ and $\mathcal{V} \neq ClausesToBeExamined$ do
          begin
            if $C_l \subseteq ClausesToBeExamined$ then
              begin
                $C_l \leftarrow C_l \cup \{CurrentClause\}$;
                $\mathcal{V} \leftarrow \mathcal{V} \cup C_l$
              end ;
            $l \leftarrow l + 1$;
          end ;
        Update $ClausesToBeExamined$ to account for those clauses that were covered by the
        cliques into which $CurrentClause$ is inserted:
        $ClausesToBeExamined \leftarrow ClausesToBeExamined \setminus \mathcal{V}$;
        while $ClausesToBeExamined \neq \{\}$ do
          begin {Add new cliques}
            Find the smallest $l$, with $1 \leq l \leq k$, such that $|C_l \cap ClausesToBeExamined|$ is maximal:
            $k \leftarrow k + 1$;
            $C_k \leftarrow (C_l \cap ClausesToBeExamined) \cup \{CurrentClause\}$;
            $ClausesToBeExamined \leftarrow ClausesToBeExamined \setminus C_l$
          end ;
        $CurrentClause \leftarrow CurrentClause + 1$
      end
  end ;
Suppose $C_1, \ldots, C_k$ are the cliques produced so far: examine them one by one to see if the
clauses covered by a clique are also covered by a subset of the union of the remaining cliques:
if a clique is subsumed by the union of the remaining cliques, then eliminate it;
$p \leftarrow$ the number of remaining cliques
```

Thus, for our particular problem, we get the following algorithm for computing $lgs\theta(\mathcal{E}, m)$:

Algorithm lgsθ
Inputs:
– a set $\mathcal{E}$ of reconcilable clauses (defining a relation r);
– a construction mode m for r.
Output:
– a non-recursive logic program $\mathcal{R}$ for r, such that $\mathcal{R} \models \mathcal{E}$.

```
$C_1, \ldots, C_p \leftarrow ClauseCliqueCover(\mathcal{E}, m)$;
$\mathcal{R} \leftarrow \{lg\theta(C_1), \ldots, lg\theta(C_p)\}$
```

Finally, the algorithm of the *Program Closing Method* is as follows:

Algorithm ProgramClosingMethod
Inputs:
– an open logic program $\mathcal{Q}$ for a relation q, but with no clauses for one used relation, say r;
– an evidence set $\mathcal{E}$ of clauses defining r;
– a construction mode m for r.
Output:
– a closed logic program $\mathcal{Q}'$ for q, such that $\mathcal{Q}' = \mathcal{Q} \cup \mathcal{R}$,
where $\mathcal{R}$ is a non-recursive logic program for r, such that $\mathcal{R} \models \mathcal{E}$.

```
$\mathcal{R} \leftarrow lgs\theta(\mathcal{E}, m)$;
$\mathcal{Q}' \leftarrow \mathcal{Q} \cup \mathcal{R}$
```

This algorithm correctly enacts all the scenarios envisaged in Examples 1 through 6, if coupled with the following acceptability criterion.

3.2 The Acceptability Criterion

The *Program Closing Method* always succeeds, but, as seen in Example 2, its result is not always acceptable. Indeed, sometimes it is necessary to reject its result and instead invoke an entire new synthesis (of a recursive program for r) from its evidence for r, precisely because the *Program Closing Method* cannot infer a recursive program for r. Such rejection plus auxiliary synthesis corresponds to *necessary predicate invention*, 'necessary' in the sense that a recursive program $\mathcal{R}$ for r cannot be eliminated by unfolding for occurrences of r in $\mathcal{Q}$. If the result $\mathcal{R}$ of the *Program Closing Method* is deemed acceptable, then relation/predicate r is 'unnecessary' in the sense that the non-recursive program $\mathcal{R}$ can be unfolded for occurrences of r in $\mathcal{Q}$. Now, it is in general undecidable whether predicate invention is necessary (via rejection) or not (via acceptance) [19]. So a heuristic is needed to judge the output of the *Program Closing Method*, and we call this heuristic the *acceptability criterion*.

Our proposed acceptability criterion is as follows. The program $\mathcal{R}$ inferred by the *Program Closing Method* is *acceptable* iff the following conditions hold:

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

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

The first condition is obvious, as it avoids over-generalization. Note that negative evidence plays *no* role in the *Progam Closing Method* per se, but it may or may not be useful in the abduction step of the basic synthesis algorithm; also, note that both the *Progam Closing Method* and the acceptability criterion can even perform 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 that uses the *Program Closing Method*. For instance, if only carefully chosen evidence is presented to the synthesis technique, say n clauses, then "not too large" could mean, say, "less than $n \div 2$". An alternative way of expressing this idea is to require that the size of each clique is larger than 2, say (remember that the lgθ of each clique is a clause of $\mathcal{R}$).

Example 7 Let $\mathcal{E}$ be the following evidence set for *insert*:

$insert(1, [\,], [1]) \leftarrow$	(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)

and let the construction mode m 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 generalizations are:

$insert(X, L, [X|L]) \leftarrow$
$insert(X, [Y|L], [Y, X|L]) \leftarrow$
$insert(X, [Y, Z|L], [Y, Z, X|L]) \leftarrow$

But this program does not fit 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 mth clause achieves insertion of a number into the mth 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 mth position of a list of numbers, where $m > 3$. This cannot be done with a finite non-recursive program (unless other predicate symbols are added), so the result of the *Program Closing Method* is inadequate and detecting this is what the acceptability criterion is for. □

4 Related Work

There are two kinds of related work. First, in Section 4.1, 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 4.2, we compare our *Program Closing Method* to other methods, which have been proposed independently of particular synthesis techniques.

4.1 Related Synthesis Techniques

This paper is a considerable extension of the second author's previous work on the SYNAPSE synthesis technique [6, 3], 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 clause completion 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 also handle clausal evidence, and hence semantic construction. The definitions of construction mode and compatibility also have undergone some changes, so the theorems all had to be re-proved accordingly.

The DIALOGS synthesis technique [5] now [23] exploits the 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.

The METAINDUCE synthesis technique [10] 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 unique *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.

The CILP synthesis technique [14] 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 data-flow; 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 data-flow. 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).

The CRUSTACEAN synthesis technique [1] is a successor of the LOPSTER technique [13], 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 also has the drawbacks of CILP.

The THESYS synthesis technique [21] is a precursor to all these techniques, but it is set in the functional programming paradigm. 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 [4]). 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, 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 has been extended, revised, and reformulated over the years as the *BMWk* technique [11, 16], and was also transposed to a higher-order logic framework [9].

Many other techniques of inductive synthesis of recursive programs, although they are not all schema-biased, are reviewed in [7].

4.2 Related Methods

The SIERES learning technique [22] 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 data-flow 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 inferability of multi-clausally defined predicates.

The INDICO learning technique [20] 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. 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.

5 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.

The existential case. 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. Such is the case when the top-level relation is non-deterministic given particular values for the chosen induction parameter and passive parameters (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 designated result parameter, but only one of them is actually used to construct (through *compose*) the result corresponding to the undecomposed induction parameter. Several such values either lead to abduced (positive) evidence with disjunction or with existentially quantified variables.

Example 8 Consider the following specification:

$firstN(N, L, R)$ iff list R is the first N elements of list L, which has at least N elements, where N is a Peano number.

If Step 2 generates the open program (using L as induction parameter):

$$\begin{array}{l} firstN(N,L,R) \leftarrow L = \_, N = 0, R = [\,] \\ firstN(N,L,R) \leftarrow L = [HL|TL], firstN(TN,TL,TR), compose_1(HL,TN,TR,N,R) \end{array}$$

then, from the evidence $firstN(s(s(0)),[a,b,c],[a,b]) \leftarrow$, at best the following (disjunctive!) evidence for $compose_1$ could be abduced (depending on the other evidence for $firstN$):

$$\begin{array}{l} compose_1(a,0,[\,],s(s(0)),[a,b]) \vee \\ compose_1(a,s(0),[b],s(s(0)),[a,b]) \vee \\ compose_1(a,s(s(0)),[b,c],s(s(0)),[a,b]) \leftarrow \end{array}$$

because there are three correct instances of the recursive call $firstN(TN,[b,c],TR)$. First of all, note that this is not a definite clause, so that the method seen here is not applicable. Also, only one of the three involved atoms for $compose_1$, namely the second one, is actually useful for proving the query $\leftarrow firstN(s(s(0)),[a,b,c],[a,b])$. Note that it also is the only one to be admissible wrt mode $compose_1(may_{all}, must_1, must_2, res_1, res_2)$.

Similarly, if Step 2 generates the open program (using N as induction parameter):

$$\begin{array}{l} firstN(N,L,R) \leftarrow N = 0, L = \_, R = [\,] \\ firstN(N,L,R) \leftarrow N = s(TN), firstN(TN,TL,TR), compose_2(TL,TR,L,R) \end{array}$$

then, from the same evidence $firstN(s(s(0)),[a,b,c],[a,b]) \leftarrow$, at best the following evidence for $compose_2$ could be abduced (depending on the other evidence for $firstN$):

$$\exists A,T \;.\; compose_2([A|T],[A],[a,b,c],[a,b]) \leftarrow$$

because $firstN(s(0),[A|T],[A])$ summarizes all the instances of the recursive call $firstN(s(0),TL,TR)$. Again, this is not a definite clause, so that the method seen here is not applicable. Also, only one instance of this evidence, namely $compose_2([b,c],[b],[a,b,c],[a,b])$, is actually useful for proving the query $\leftarrow firstN(s(s(0)),[a,b,c],[a,b])$. Note that it also is the only one to be admissible wrt mode $compose_2(must_1, must_2, res_1, res_2)$. □

In a forthcoming paper, we will show how to handle this existential case (for an old version of this solution, consult [3]). Essentially, the useless components of the evidence, respectively the useless instances of the evidence, have to be eliminated, and the admissibility criterion plays a crucial role here: the more precise it is, the better the results of this elimination process.

Recursive evidence. As of now, the *Program Closing Method* is restricted to abduced evidence in the form of non-recursive (proper) clauses. There is no real 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 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 3 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 [5, 23]. 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 model 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 [5, 23]), or by extraction from the evidence for the top-level relation (as in SYNAPSE [6, 3]), or by some form of "background knowledge usage miracle" [7] (as in the vast majority of inductive synthesizers). (Also remember that θ-subsumption suffices for non-recursive clauses, which is the case 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 model of generality in order to "push" the mentioned assumption inside the *Program Closing Method.*

6 Conclusion

We have given a new definition of the concept of least generalization (under θ-subsumption) of a set $\mathcal{C}$ of clauses, denoted by $lg\theta(\mathcal{C})$. The 'classical' approach is to define $lg\theta(\mathcal{C})$ as a *single* clause. Since such a unique clause is sometimes too general, we have here re-defined $lg\theta(\mathcal{C})$ as being a *set* of clauses, each member of this set being the least generalization (under 'classical' θ-subsumption) of some subset of $\mathcal{C}$. The considered subsets have $\mathcal{C}$ as their union, but do not necessarily constitute a partition of $\mathcal{C}$. These subsets are here called compatible sets, because their elements are two by two compatible, in the sense that their least generalizations (under 'classical' θ-subsumption) are not too general. Moreover, these subsets must each be of maximal size such that their least generalizations (under 'classical' θ-subsumption) are not too general. Or, in other words, the number of these subsets must be minimal.

The criterion for over-generality is of course problem-specific. We have here designed such a criterion for a problem frequently occurring in the inductive synthesis of recursive logic programs, namely the completion (or: closing) of an open program that was synthesized in a schema-biased way, but that has one of the place-holders of the used schema still undefined. Since a schema captures the data-flow of all programs designed by a certain methodology, it also captures the data-flow of the undefined place-holder, and it is this data-flow that gives rise to the over-generality criterion for this problem: a least generalization (under θ-subsumption) is over-general if it is not admissible wrt a certain construction mode. We have designed a language for expressing such construction modes (for any place-holder of any schema), and we have defined a powerful admissibility criterion. We have also proposed a few theorems relating the problem-independent concept of compatibility with the problem-specific concept of admissibility: these theorems show how to speed up certain computations for this specific problem.

Let us now return the discussion to the general problem. We have shown an algorithm that computes the least generalization (under θ-subsumption) of a set of clauses, according to our new definition. This amounts to (1) finding the node clique cover of an undirected graph, whose nodes are the given clauses and whose edges represent the compatibility relationships among these clauses; and (2) computing the least generalizations (under θ-subsumption) of these cliques, according to the 'classical' definition. This algorithm is thus fully general, and may be used, with an appropriate over-generality criterion, for any particular instance of the problem that the reader may encounter.

Acknowledgments

We thank Baudouin Le Charlier and Pierre-Yves Schobbens (both at the University of Namur, Belgium) for some insightful remarks during a presentation of a preliminary version of the method presented here. We are also grateful to the Machine Learning Group at the University of Texas at Austin, for their helpful comments during a discussion of 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, Bilkent University, Ankara (Turkey), May 1996.

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

[4] P. Flener. *Predicate invention in inductive program synthesis.* Technical Report BU-CEIS-9509, Bilkent University, Ankara (Turkey), 1995.

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

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

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

[8] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[9] 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.

[10] 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.

[11] 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.

[12] T. Kou, L.J. Stockmeyer, and C.K. Wong. Covering edges by cliques with regard to keyword conflicts and intersection graphs. *Comm. of the ACM* 21(2):135–139, Feb. 1978.

[13] 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.

[14] 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.

[15] K.-K. Lau and M. Ornaghi. The relationship between logic programs and specifications: The subset example revisited. *J. of Logic Programming* 30(3):239–257, 1997.

[16] 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.

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

[18] 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.

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

[20] 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.

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

[22] 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.

[23] S. Yılmaz. *Inductive Synthesis of Recursive Logic Programs.* M.Sc. thesis. Technical Report BU-CEIS-9717, Bilkent University, Ankara (Turkey), 1997.