A Re-Definition of Least Generalizations, and Construction Modes as a New Declarative Bias for ILP

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 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 classical least generalization of some subset of $\mathcal{C}$. The elements of these subsets are two by two *compatible*, in the sense that their classical least generalizations are not too general according to some over-generality criterion. We give an algorithm for computing this re-defined concept, providing thus a new approach to disjunctive concept learning.

The criterion for over-generality is problem-specific. We introduce a new such criterion: two clauses are compatible, i.e., their classical least generalization is not over-general, if this least generalization is *admissible* wrt a *construction mode* capturing the required dataflow of each involved relation. We design a language for expressing such construction modes, and we define a powerful admissibility criterion. We also prove a few theorems relating the problem-independent concept of compatibility with the problem-specific concept of admissibility (our over-generality criterion): these theorems show how to speed up certain computations for these concepts. Finally, we outline an application of these considerations to a frequently occurring problem in the inductive synthesis of recursive (logic) programs, namely the induction of the operator combining the partial results (obtained through recursion) into the overall result.

1 Introduction

The classical definition of the concept of least general generalization (or simply: least generalization) of a clause set is that it is a *single* clause. We first repeat this definition and then exhibit one of its problems. For the sake of illustration, a simple generality model will suffice in this paper, namely θ-subsumption [16]. Our ideas are indeed independent of the chosen generality model.

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 model of generality for clauses [16]. 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 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 $\sigma = \{I/X, L/[HL|TL]\}$.

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 [16]:

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.[1]

This is what we here call the classical definition of the lgθ. One of its problems that is not due to the choice of θ-subsumption as underlying generality model (remember that we chose this simple model just for the sake of illustrating some new ideas) is that the *single* clause that θ-subsumes all the clauses in the set $\mathcal{C}$ is often too general.

Example 1 For instance, take the following clause set:

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

Its lgθ is $combine(X, T, V) \leftarrow$, which is over-general if *combine* is meant to fit into the program:

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

where $delOddElems(L, R)$ holds iff list R is integer-list L without its odd elements. In fact, the *combine* relation should be defined as follows:

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

This is a conjunctive definition, or a disjunctive one, if one writes it as $combine(I, L, R) \leftrightarrow (odd(I) \wedge R = L) \vee (even(I) \wedge R = [I|L])$. We are thus in the context of disjunctive predicate learning. □

Note that each of these two clauses is the classical lgθ of a subset of the given clause set. Hence the idea of re-defining the notion of least generalization (under θ-subsumption) of a clause set $\mathcal{C}$ as being a minimal-sized *set* of n clauses c_i, where $n \leq |\mathcal{C}|$, 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., $\bigcup_{i=1}^{n} \mathcal{C}_i$ must form a *cover* of $\mathcal{C}$). The elements of these subsets must be two by two *compatible*,

[1] 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.

in the sense that their classical least generalizations (under θ-subsumption) are not too general, according to some over-generality criterion. 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 the over-generality criterion) 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 cover is a partition and we look for a minimal cover (i.e., a cover with the smallest number of subsets). We will give a generic re-definition (wrt any over-generality criterion) of the least generalization (under θ-subsumption) of a clause set, and present an efficient algorithm for computing it.

The criterion for over-generality is problem-specific. We will also introduce a new such criterion: two clauses are compatible, i.e., their classical least generalization is not over-general, if this least generalization is *admissible* wrt a *construction mode* capturing the required dataflow of each relation. We will design a language for expressing such construction modes, and we will define a powerful admissibility criterion based on it. We will 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 these concepts.

In the rest of this paper, we thus first define, in Section 2 through Section 4 respectively, the notions of construction mode, admissibility, and compatibility. Then, in Section 5, we give the re-definition for the least generalization of a clause set, as well as an efficient algorithm for computing it. In Section 6, we discuss an interesting application of the presented material, for a frequently occurring problem in the inductive synthesis of recursive (logic) programs, namely the induction of the operator combining the partial results (obtained through recursion) into the overall result. Finally, in Section 7, we review related work, outline future work, and conclude.

2 Construction Modes: A New Declarative Bias

Informally, a construction mode 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. We now incrementally define the notion of construction mode.

2.1 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 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 $vertices(a \cdot T) = \{a, \cdot, T\}$.

Definition 4 (Syntactic construction)
Term s is *syntactically obtained from* term t if $leaves(t) \subseteq leaves(s)$. We denote this by $t \subseteq s$.
Term s *syntactically contains* term t if $vertices(t) \sqsubseteq vertices(s)$, 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 between their parameters: this will be the role of construction modes (defined below).

The reason why we sometimes consider functions 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 (rather than just 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 $efface(d, [g, e, d], [g, e])$, 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 $[g, e, d]$.

2.2 Semantic Construction

To capture more than just syntactic construction (which is basically achieved using unification only, inside a single atom), we have to extend this notion to semantic construction over definite clauses. Indeed, body atoms may perform some computations of semantic construction of some parameters in the head atom, using relations other than equality. Such atoms cannot be evaluated into the head of the clause, unlike equality atoms. For instance, in $min(X, Y, Z) \leftarrow X \leq Y, Z = X$, one can evaluate $Z = X$ into the head $min(X, Y, Z)$, yielding $min(X, Y, X) \leftarrow X \leq Y$, but one then cannot further evaluate $X \leq Y$ into $min(X, Y, X)$. Also, parameter Y does not syntactically contribute to constructing result X (the third parameter), but it does so semantically (via $X \leq Y$).

2.3 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 $\{may_1, \ldots, may_n, may_*, must_1, \ldots, must_n, res_1, \ldots, res_n\}$, such that res_j is in the range of m iff may_j or $must_j$ is in the range of m, and such that every res_j is at most once in the range of m. We say that $m(i)$ is the *mode* of the i $^{\text{th}}$ parameter of r.

A construction mode m for r is here 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 modes in this paper, we often simply speak about modes here.

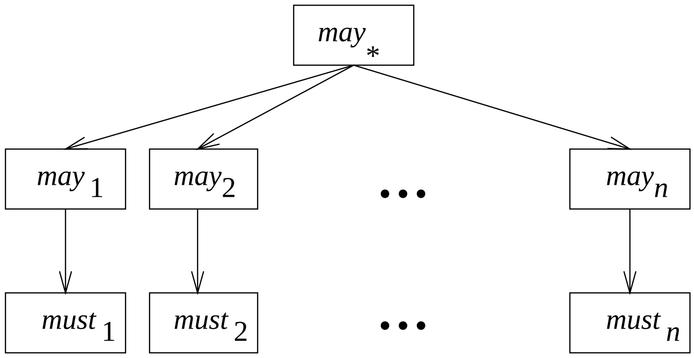
For instance, $combine(must_1, must_1, res_1)$ is a mode, meant to express that the third parameter is a result parameter that must be constructed from the first two parameters. For convenience, we often drop the indexes j of may_j, $must_j$, and res_j when they all have the same value. So the mode above can also be written as $combine(must, must, res)$. Also, $r(may_*, must_1, must_2, res_1, res_2)$ is a mode, meant to express that the fourth parameter is a result parameter that must be constructed from the second parameter and may be constructed from the first parameter, and that the fifth parameter is a result parameter that must be constructed from the third parameter and may be constructed from the first parameter. Note that $r(may_1, must_1)$ is not a mode, because no parameter is being designated as the result of construction from the two given parameters. Also, $r(res_1)$ is not a mode, because it does not indicate from what parameters the given result parameter must or may be constructed. Moreover, $r(must_1, res_1, res_1)$ is not a mode, because a result parameter cannot possibly be in two places at the same time. Finally, $r(may_*, may_*)$ is a mode.

In a first approximation (for syntactic construction), the intended semantics of a construction mode is thus as follows:

- mode res_j means the parameter in the corresponding position is a *result parameter*, numbered j, to be constructed from other (non-result) parameters;
- 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_* means the parameter in the corresponding position is optional for syntactically constructing any of the parameters in the corresponding positions of all res_j.

We will refine (for semantic construction) and formalize all this in the following, via the concept of admissibility. Note that mode may_* generalizes every may_j, and that each may_j itself generalizes $must_j$. The following figure illustrates this partial generality order:



So the mode $reverse(must_1, res_1)$ could be rewritten as $reverse(may_1, res_1)$, but the ensuing loss of knowledge might damage the precision of any computations based on modes. It is thus always preferable to use a least general construction mode, according to the partial order of the figure above. It may not be unique, though: for instance, $reverse(res_1, must_1)$ is an alternative mode.

Some relations do not have any result parameters. For instance, $even(N)$ and $X \leq Y$ are just tests; this is accommodated here by just setting their modes to $even(may_*)$ and $\leq (may_*, may_*)$.

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). Also, a *not* mode could be introduced, expressing that the parameter in the corresponding position may not be used for constructing any of the result parameters. We do not consider such extensions in this introductory paper, but the corresponding generalizations are straightforward. Our objective here is merely to establish some simple concepts. The key issue is that modes can be pre-determined for any relation.

3 Admissibility

In a first version, the concept of admissibility captures 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 it 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.

3.1 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 $Must = \langle t_i \mid m(i) = must_j \text{ for some } j \rangle$. Similarly for May_j, May_*, May, Res_j, and Res.

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

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

For instance, this is the case for the r atom and mode above. Note that k may be 0, such as in $\leq (may_*, may_*)$; 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_* 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 the parameters in the corresponding positions of may_j, may_*, and $must_j$, then we can formalize this as follows:

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

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

For instance, the atom $addOne(a, 0, s(0))$ satisfies condition (2′) for $addOne(may_1, must_1, res_1)$ 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. A constructor constant of the inductively defined type of such a parameter may appear: for instance, 0 is such a constant for Peano numbers, and nil for lists. The atom $addPlateau(a, [\,], [a, s(0)])$ does not satisfy condition (2′) for $addPlateau(may_1, must_1, res_1)$, because constant 0 is "invented" by the parameter with mode res_1. Since such constructor constants cannot be considered new if the inductively defined type is known, we should add them to the right-hand side:

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

We here do not allow non-constructor constant leaves to be invented by res_j parameters, and leave such extensions as future work. Hence the following definition:

Definition 6 (Atom admissibility)
An atom $r(t_1, \ldots, t_n)$ is *admissible* wrt a mode m for r if conditions (1) and (2) above are satisfied.

3.2 Semantic Admissibility of a Clause wrt a Construction Mode

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 7 (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.

For defining the semantic admissibility of a proper clause wrt a construction mode, we have to distinguish between its head atom and its body atoms. For the head atom, we first want its *must* parameters to be actually used in constructing the result parameters. Condition (1) only verifies syntactic construction, but some of the leaves of the *must* parameters might only be used in the body $\mathcal{B}$ to achieve semantic construction. So condition (1) must be 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 $\mathcal{B}$. Second, we want the result parameters to be constructed only from the *must* and *may* parameters, as well as from the predefined constructor constants. Condition (2) only verifies syntactic construction, but some of the leaves of the result parameters might only be computed in the body $\mathcal{B}$, by semantic construction thus. So condition (2) must be adapted as follows:

$$\forall 1 \leq j \leq k : Res_j \subseteq \langle May_j, May_*, Must_j, \mathcal{B}', 0, nil, \ldots \rangle \tag{4}$$

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

Conditions (3) and (4) can certainly be refined even further, say via a notion of linkedness of the vertices of the involved terms, but we leave this for future work.

For the body atoms now, other than their participation in conditions (3) and (4) above, it is necessary to verify whether they are each admissible (according to the previous definition) wrt their own construction modes. This only establishes their syntactic admissibility, but there is nothing else that can be done since they are not the head atoms of proper clauses.

Now we can finally propose the following definition of clause admissibility:

Definition 8 (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 conditions (3) and (4) above are satisfied, and if the atoms of $\mathcal{B}$ are each admissible wrt their own modes.
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.

Admissibility has some interesting properties, as established next. They only hold for (sets of) proper clauses whose bodies only involve atoms for test relations (whose modes only involve the may_* mode), so that their body atoms are all trivially admissible.

Lemma 1 (Preservation of admissibility under θ-subsumption)
If proper clause c is admissible wrt a 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 literal sets). Supposing c has the structure $r(t) \leftarrow \mathcal{B}$, for some tuple t and body $\mathcal{B}$, this 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, the clause $c\sigma$ is also admissible wrt m, by the rule of universal instantiation. 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. $\Box$

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 1 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. $\Box$

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 $insert(must, must, res)$, but its least generalization $insert(X, [Y], [K, M]) \leftarrow$ is not admissible wrt that mode.

4 Compatibility and Over-Generality

Compatibility is a problem-independent concept that will be used in our re-definition of least generalizations. Compatibility is relative to an over-generality criterion, which is problem-specific.

Definition 9 (Over-generality criteria)
An *over-generality criterion* is a boolean function over the set of definite clauses.

For instance, admissibility of a proper clause wrt a construction mode for the relation in its head is an over-generality criterion. Another criterion could be the non-coverage of any given negative examples, or even the combination of these two criteria.

Intuitively, compatibility of two clauses is meant to capture the fact that these two clauses perform something 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 same thing 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. Hence the following definition:

Definition 10 (Clause compatibility)
Two reconcilable clauses c and d are *compatible (with each other)* wrt an over-generality criterion G if $G(lg\theta(c,d))$ holds. We also say that c is *compatible with* d wrt G, 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 admissibility wrt $combine(may, must, res)$, because their lgθ, namely $combine(I,L,[I|L]) \leftarrow even(I)$, is admissible wrt that mode.

When a mode m or an over-generality criterion G has been clearly stated in context, we often drop the qualifiers "wrt mode m" (for admissibility) and "wrt criterion G" (for compatibility). If the chosen over-generality criterion is admissibility wrt some mode m, then we often say "compatible wrt m" instead of "compatible wrt admissibility wrt m".

In order to be completely general, we do not impose any constraints on the compatibility relation. It is symmetric (by construction), but we do not require it to be reflexive nor transitive.

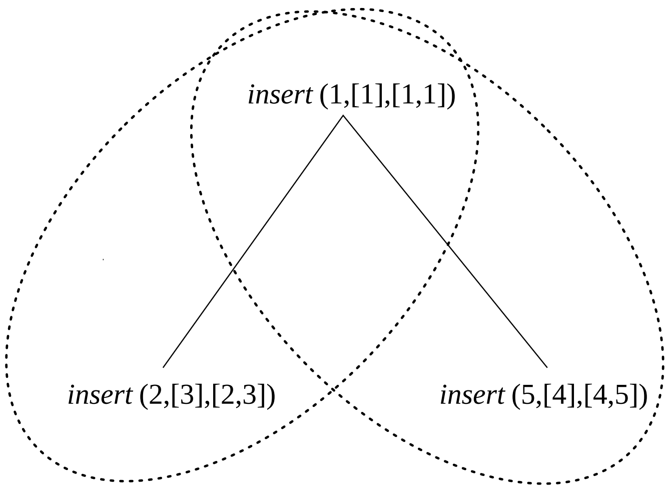
For instance, compatibility wrt admissibility is not reflexive, because the definition does not require the two given clauses to be admissible wrt the given mode. For instance, the clause $q(d,[g],[e,g]) \leftarrow$ is not admissible wrt $q(may, must, res)$, therefore it is not compatible with itself (because its lgθ with itself is itself). In general, reflexivity is really irrelevant for our purposes.

Also, compatibility wrt admissibility is not transitive. For instance, the clauses $insert(1,[1],[1,1]) \leftarrow$ and $insert(2,[3],[2,3]) \leftarrow$ are compatible wrt admissibility 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 often impossible to achieve without sacrificing most of the power of our re-definition of least generalizations. Indeed, if compatibility were to be transitive, then some clauses would have to be compatible (e.g., the last two clauses above), although they do not construct their res_j parameters in the same 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 11 (Clause set compatibility)
A non-empty set $\mathcal{C}$ of reconcilable clauses is *compatible* wrt an over-generality criterion G if any two clauses in $\mathcal{C}$ are compatible wrt G.

Note that a singleton set is thus trivially compatible wrt *any* over-generality criterion. Also notice that a compatible subset of a set $\mathcal{D}$ of reconcilable clauses is a clique of the graph with node set $\mathcal{D}$ and edge set induced by the compatibility relationship over $\mathcal{D}$. Remember that a *clique* (or: *complete sub-graph*) of a graph $\langle\mathcal{N},\mathcal{E}\rangle$ is a subset $\mathcal{N}'$ of node set $\mathcal{N}$ such that any two nodes in $\mathcal{N}'$ are joined by an edge in edge set $\mathcal{E}$. 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 wrt admissibility wrt $insert(must, must, res)$:



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

Note that, due to the non-required reflexivity and transitivity of compatibility, cliques are not simply the equivalence classes of an equivalence relation: this considerably complicates clique finding, as seen in Section 5.

We now establish some interesting properties of compatibility wrt admissibility. They also only hold for (sets of) proper clauses whose bodies only involve atoms for test relations.

Checking whether a given set is compatible wrt admissibility seems to be an extremely tedious 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 over-generality criterion, so that no decision procedure is needed then).

Theorem 2 (Sufficient criterion for clause set compatibility wrt admissibility)
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 10 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 11. □

Note the following useful property linking the two $lg\theta$ operators:

Property 3 If $c \in \mathcal{C}$, then $lg\theta(lg\theta(\mathcal{C}),c) = lg\theta(\mathcal{C})$.

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 11 and by the assumption)
iff $lg\theta(lg\theta(\mathcal{C}),c)$ is admissible wrt m (by Definition 10)
iff $lg\theta(\mathcal{C})$ is admissible wrt m (by $c \in \mathcal{C}$ and Property 3)
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). $\square$

This basically means that the lgθ of a compatible set can be taken as a representative of that set, because it is compatible with *each* element in that set.

5 The Least Generalization of a Clause Set Revisited

Given a set $\mathcal{C}$ of definite clauses and an over-generality criterion G, we now want to define a (new) concept of least generalization of $\mathcal{C}$ wrt G, such that it does not suffer from the mentioned drawback of the classical definition. Basically, we want a *set* $\mathcal{L}$ of clauses as the least generalization. The idea of the re-definition comes from Theorem 3: one can divide $\mathcal{C}$ into a minimal number of subsets that are compatible wrt G, and then take the set of their classical least generalizations as representatives, i.e., as $\mathcal{L}$.

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

Example 2 Let $\mathcal{C}$ be the clause set of Example 1:

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

The entire set $\mathcal{C}$ is not compatible wrt $combine(must, must, res)$, because, for instance, clauses E_1 and E_4 are not compatible, since $lg\theta(E_1, E_4)$, namely $combine(X,T,V) \leftarrow$, is not admissible. So we need a division of $\mathcal{C}$ into a minimal number of subsets such that each one is compatible. In this case, the subsets $\mathcal{C}_1 = \{E_1, E_3\}$ and $\mathcal{C}_2 = \{E_2, E_4\}$ are compatible, because both $lg\theta(E_1, E_3)$, which is $combine(X,T,T) \leftarrow odd(X)$, and $lg\theta(E_2, E_4)$, which is $combine(X,T,[X|T]) \leftarrow even(X)$, are admissible. Therefore, we get the following least generalization:

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

which is a partially correct and complete program for *combine*. $\square$

Example 3 Let $\mathcal{I}$ be the following clause set for *insert*:

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

The entire set $\mathcal{I}$ is not compatible wrt $insert(must, must, res)$, because, for instance, clauses E_2 and E_3 are not compatible, since $lg\theta(E_2, E_3)$, namely $insert(X,[Y],[Z,T]) \leftarrow$, is not admissible. So we need a division of $\mathcal{I}$ into a minimal number of subsets such that each one is compatible. In this case, the subsets $\mathcal{I}_1 = \{E_1, E_2\}$ and $\mathcal{I}_2 = \{E_1, E_3\}$ are compatible, because both $lg\theta(E_1, E_2)$, which is $insert(X,[Y],[X,Y]) \leftarrow$, and $lg\theta(E_1, E_3)$, which is $insert(X,[Y],[Y,X]) \leftarrow$, are admissible. Therefore, we get the following least generalization:

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

which is a partially correct (but incomplete) program for *insert*. $\square$

In these two examples, while we are trying to find some subsets of the given clause 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 clauses. Note that $\mathcal{I}_1$ and $\mathcal{I}_2$ do not form a partition in Example 3, while $\mathcal{C}_1$ and $\mathcal{C}_2$ do form a partition in Example 2. Actually, in Example 3, we could have used a partition such as $\mathcal{I}'_1 = \{E_2\}$ and $\mathcal{I}'_2 = \{E_1, E_3\}$, but a cover such as $\mathcal{I}_1 = \{E_1, E_2\}$ and $\mathcal{I}_2 = \{E_1, E_3\}$ is better, because the more evidence in a subset, the more accurate its least generalization is. Therefore, in general now, what we try to find is a *node clique cover* (NCC) of the compatibility graph of the given clause set wrt a given over-generality criterion. 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 over-generality criterion over the clause 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 introduce our re-definition of the concept of least generalization of a set of clauses:

Definition 12 (Least generalization, under θ-subsumption, of a clause set, wrt an over-generality criterion)
The *least generalization under θ-subsumption* of a non-empty set $\mathcal{C}$ of clauses wrt an over-generality criterion G, denoted by $lg\theta(\mathcal{C}, G)$, 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 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 [9]:

- 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 was very inefficient in time, 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 instead of nodes, and about compatibility (wrt an over-generality criterion G) 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 (proposed by Kellerman) 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 G, and labeled j, where $j < i$. This heuristic is improved in [12], adding one additional pass over all of the cliques to eliminate the cliques subsumed by the others. 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.

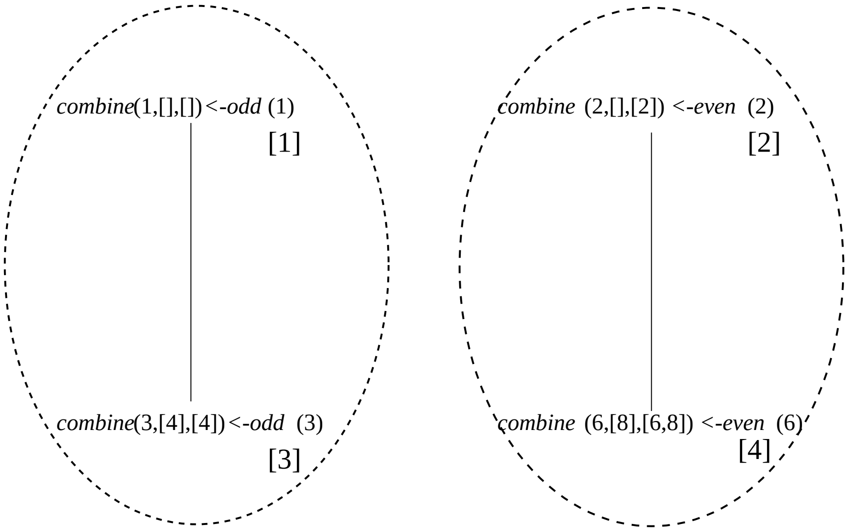
Algorithm $lg\theta$
Inputs:
– a set $\mathcal{C}$ of n clauses, whose elements are labeled from 1 to n
– an over-generality criterion G
Output:

– $lg\theta(\mathcal{C}, G)$
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
  $ClausesToBeExamined \leftarrow \{j | j < CurrentClause \wedge compatible(CurrentClause, j, G)\}$;
  if $ClausesToBeExamined = \{\}$ **then**
   begin {Create a new clique}
    $k \leftarrow k + 1$;
    $C_k \leftarrow \{CurrentClause\}$;
    $CurrentClause \leftarrow CurrentClause + 1$
   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
return $\{lg\theta(C_1), \ldots, lg\theta(C_p)\}$

Example 4 Let us analyze how this algorithm works from the clause set $\mathcal{C}$ of Example 2:

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

Let $combine(must, must, res)$ be the construction mode. Initially, all clauses in $\mathcal{C}$ are labeled starting from 1 as in the figure below:



At the beginning, there is no clique. We traverse all of the clauses in label order in order to check whether any of them is already in a clique or requires a new clique to be created.

First, we examine clause 1: as there are no clauses labeled less than 1, there is no clique that clause 1 can be inserted into. Therefore, we create a new clique $\mathcal{C}_1 = \{1\}$.

Next, we examine clause 2: there is one clause labeled less than 2, namely clause 1. However, clause 1 is not compatible with clause 2, so we cannot insert clause 2 into an existing clique. Hence, we create a new clique $\mathcal{C}_2 = \{2\}$.

Then, we examine clause 3: there are two clauses labeled less than 3, namely clauses 1 and 2. However, only clause 1 is compatible with clause 3. As all clauses labeled less than 3 are already in some existing cliques, if any of them is compatible with clause 3, then clause 3 can be inserted into a clique as well. Therefore, we eliminate clause 2 as a candidate, as it is not compatible with clause 3, and we try to insert clause 3 into existing cliques that cover clause 1, as clause 1 is compatible with clause 3. Hence, we insert clause 3 into $\mathcal{C}_1$, giving $\mathcal{C}_1 = \{1, 3\}$.

Finally, we examine clause 4: there are three clauses labeled less than 4, but only clause 2 is compatible with clause 4. Therefore, we try to insert clause 4 into existing cliques that cover clause 2; so we insert clause 4 into $\mathcal{C}_2$, giving $\mathcal{C}_2 = \{2, 4\}$.

As there is no other clause to be examined, and as neither $\mathcal{C}_1$ nor $\mathcal{C}_2$ is unnecessary to form a clique cover, the final cliques are $\mathcal{C}_1 = \{1, 3\}$ and $\mathcal{C}_2 = \{2, 4\}$. The final result is the set of classical $lg\theta$s of these cliques:

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

which is indeed the clause set aimed at in Example 1. □

Both the heuristic proposed by Kellerman and the suggested improvement of this heuristic are polynomial-time [12]. Note that the outer while-loop implements the heuristic proposed by Kellerman, and that the last step implements the improvement over this heuristic. Hence, this approximation algorithm is polynomial-time.

We have implemented this algorithm in Sicstus Prolog. We ran some experiments comparing the time efficiencies of this algorithm and a brute-force one (which is also implemented in Sicstus Prolog [2]). In the brute-force algorithm, first the set $\mathcal{S}$ of all complete subgraphs is found, and then a minimal set $\mathcal{C}$ among the subsets of $\mathcal{S}$ such that $\mathcal{C}$ forms a clique cover is produced as a candidate minimal clique cover. The subsets of $\mathcal{S}$ with smaller cardinalities are examined first, so as to improve the efficiency of the algorithm. Although the brute-force algorithm can produce all the minimal clique covers, for the experimental results we only took into account the first minimal cover produced. The experiments were run on a Sun Sparc station. Our data set consisted of graphs of at most 5 nodes and at most 10 edges. Only one random configuration of a graph of n nodes and e edges is included in the data set. In the graphs of the data set, the clauses (i.e., the nodes) are represented by integers, and the compatibility relation (i.e., the edges) is represented by an adjacency relation. For each graph, the algorithms are run 10 times and the average CPU times are found in milli-seconds (ms). The results are summarized in the following table:

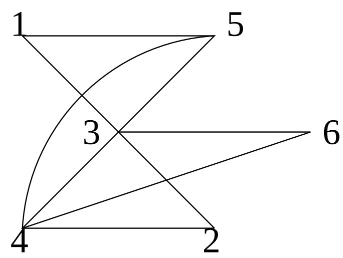
# of nodes	# of edges	# of cliques	time (ms) brute-force	time (ms) approximation
2	0	2	1	0
2	1	1	2	1
3	0	3	3	0
3	1	2	2	0
3	2	2	2	1
3	3	1	3	0
4	0	4	5	1
4	1	3	5	1
4	2	3	7	1
4	3	3	9	0
4	4	2	7	1
4	5	2	9	0
4	6	1	10	1
5	0	5	11	2
5	1	4	12	2
5	2	4	15	2
5	3	3	12	2
5	4	3	14	1
5	5	3	19	2
5	6	3	23	2
5	7	3	32	2
5	8	2	19	2
5	9	2	21	1
5	10	1	29	1

In both algorithms, the time of finding a minimal cover is proportional to the size of the graph (i.e., the numbers of nodes and edges) and its configuration (i.e., the number of cliques). In both algorithms, if we consider the graphs with n nodes, then even if the number of edges increases, the time taken to find a minimal clique cover may decrease, because the number of cliques may decrease. In the brute-force algorithm, this is due to the fact that the smaller subsets of the set of all complete subgraphs are examined before the others. In the approximation algorithm, this is due to the incremental construction of the cliques of a minimal clique cover.

It is not surprising that the brute-force algorithm takes much more time than the approximation algorithm, because the brute-force algorithm is exponential-time (as all of the complete subgraphs have to be enumerated before finding a minimal clique cover), while the approximation algorithm is polynomial-time (as the cliques are built incrementally without an exhaustive search).

As for accuracy, the approximation algorithm works well for small problems, that is, the ratio of the number of cliques found by this algorithm to the optimal number of cliques is small if the graphs consist of few nodes and edges. However, this ratio gets large for some large problems like the one constructed in [12]. This ratio is proven to be bounded [9, 12], that is, it seems unlikely that there is a polynomial-time algorithm for the NCC problem with a ratio less than 2.

We also applied both algorithms to the following graph:



The brute-force algorithm found $C_1 = \{3, 4, 6\}$, $C_2 = \{2, 3, 4\}$, and $C_3 = \{1, 3, 5\}$ as a minimal clique cover in 197 milli-seconds (as an average over 10 trials). Note that the brute-force algorithm can find all minimal covers but that we consider only the first one produced. On the other hand, the approximation algorithm only finds one of the minimal clique covers. The approximation algorithm found $C_1 = \{5, 3, 1\}$, $C_2 = \{4, 3, 2\}$, and $C_3 = \{6, 4, 3\}$ as a minimal clique cover in less than 1 milli-second. The minimal clique cover that the brute-force algorithm finds has maximal-sized cliques because the brute-force algorithm examines the minimal clique covers with maximal-sized cliques before the others. On the other hand, in the approximation algorithm the cliques of a minimal clique cover are built incrementally. Note that if we had not inserted nodes into as many cliques as possible (in the first inner while-loop), then we would have ended up with four cliques (namely $C_1 = \{1, 5\}$, $C_2 = \{2, 3, 4\}$, $C_3 = \{3, 4, 5\}$, and $C_4 = \{3, 4, 6\}$) as a minimal clique cover: at the time we examine node 3, the available cliques are $C_1 = \{1\}$ and $C_2 = \{2\}$; if we had inserted node 3 into only C_2 (rather than into both available cliques) in the first inner while-loop, then, at the time we examine node 5, the available cliques would be $C_1 = \{1\}$ and $C_2 = \{2, 3, 4\}$; we would insert node 5 into C_1 (in the first inner while-loop), and also add a new clique (in the second inner while-loop), i.e., $C_3 = \{3, 4, 5\}$; therefore, at the time we examine node 6, the available cliques are $C_1 = \{1, 5\}$, $C_2 = \{2, 3, 4\}$, and $C_3 = \{3, 4, 5\}$; hence, four cliques will be available as a minimal clique cover at the end, namely $C_1 = \{1, 5\}$, $C_2 = \{2, 3, 4\}$, $C_3 = \{3, 4, 5\}$, and $C_4 = \{3, 4, 6\}$.

As a result of these experiments, we can see that this approximation algorithm is more efficient in time than the brute-force one, but it is less efficient in accuracy than the brute-force one as the approximation algorithm is not guaranteed to find a minimal clique cover.

6 An Application: The Program Closing Method

In a companion paper [3], we apply construction modes, admissibility, and the here re-defined notion of least generalization (wrt admissibility) of a clause set in order to provide a novel solution to part of the problem of inductive synthesis of recursive (logic) programs from incomplete specifications. Indeed, every now and then, inductive synthesizers appear, having the following *basic synthesis algorithm* [7], given clausal evidence for a top-level relation r:

1. *Schema-biased* [2] *creation* of an open [8] 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 [17] 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.

[2] 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 [8].

Some synthesizers of this category are THESYS [19], *BMWk* [11, 15], SYNAPSE [6, 4], LOPSTER [13], CILP [14], CRUSTACEAN [1], METAINDUCE [10], DIALOGS [5, 7], etc. We here ignore the mechanics of Steps 1 and 2: there are various ways of achieving the results reported here (or similar ones) and we invite the reader to accept them as such, because our focus is on Step 3. Let us illustrate all this on an example.

Example 5 Take the informal specification: $delOddElems(L, R)$ holds iff list R is integer-list L without its odd elements. A specifier could give the following (formal) specification by incomplete clausal evidence:

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

Suppose 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 abduces the following clausal 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 of the THESYS, *BMWk*, LOPSTER, CILP, CRUSTACEAN, and METAINDUCE systems induces $combine(X, T, V) \leftarrow$ as generalization of the positive evidence (usually by taking the classical lgθ), which is not an appropriate result, as argued in Example 1. In fact, the *combine* relation should be conjunctively defined as follows:

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

In general, these systems can only infer single unit clauses from abduced positive and negative examples of the open relation, rather than sets of definite clauses (with bodies, thus) from clausal evidence of the open relation. The SYNAPSE and DIALOGS systems can however infer the correct generalization above, since their Step 3 is based on the *Program Completion Method* [3], which is itself based on construction modes, admissibility, and the here re-defined notion of least generalization (wrt admissibility). Other features under which our *Program Completion Method* improves on the methods of these other systems are discussed in the companion paper [3]. □

7 Conclusion

We have introduced construction modes as a new declarative bias to ILP. A construction mode captures the required dataflow of a relation, by expressing which parameters are constructed from which other parameters, as well as whether such construction is mandatory or optional for each participating parameter. The semantics of a construction mode is formalized by the notion of admissibility of a definite clause wrt that mode, capturing both syntactic and semantic ways of term construction. Independently of all this, we have argued for a re-definition of the notion of least generalization of a clause set, from a single clause to a set of clauses, the clinching factor being the introduction of an over-generality criterion. Joining the two strands of ideas now, we have briefly shown, on a particular application, that admissibility wrt a construction mode is a very useful over-generality criterion when computing least generalizations.

We want to stress once again that our re-definition of the notion of least generalization of a clause set is essentially independent of the chosen generality model. We just chose the simple model of θ-subsumption for illustration purposes, and explored it in some depth by deriving some useful theorems for it. A similar exercise can be done for other, more powerful generality models that may or may not be relative background knowledge, such as inverse resolution, inverse implication, etc. Note that the given algorithm for computing this new concept can be straightforwardly adapted for any other generality model: it suffices to change the last line accordingly and to re-define compatibility accordingly.

Similarly for the over-generality criterion: the mentioned admissibility wrt a construction mode is just one possibility, introduced for the sake of being able to show an application, but the reader may use the new least generalization algorithm with any over-generality criterion. For instance, non-coverage of negative examples is a commonly used such criterion.

It should be noticed that we have taken a very general position as to what constitutes evidence to be generalized: indeed, we consider definite clauses as evidence. It is obvious that the (old-fashioned) approach of evidence as positive and negative literals (or examples) is covered by ours, as it suffices to view positive examples as unit clauses and negative examples as head-less clauses with one body atom.

We do not want to present our new least generalization algorithm as one of inductive learning/synthesis, because it may not "solve" the ILP problem, or the disjunctive predicate learning problem, by itself, but only as an auxiliary to more powerful and intricate techniques. The given application inside our Program Closing Method for inductive synthesis of recursive (logic) programs is within such a highly suitable situation, because the open relations there are always of the kind where some parameters are indeed mandatorily constructed from some parameters and optionally from others, and such that this is even known in advance (by virtue of pre-determination based on the underlying program schema).

In terms of related work, there does not seem to be much, other than our own already mentioned previous work. The INDICO learning technique [18] 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 classical 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 algorithm.

In terms of future work, we have already mentioned interesting extensions to the construction mode language (see Section 2) and the definition of admissibility (see Section 3). As far as compatibility and the new definition of least generalizations are concerned, our results are totally generic, so that any combination of generality model and over-generality criterion can be used.

Of course, (refinements of) our suggested construction modes may be used (maybe in conjunction with input modes) by other researchers, even without their having to use the new least generalization definition and algorithm, and we look forward to seeing such applications of our proposal in the ILP literature.

Acknowledgments

We thank Baudouin Le Charlier and Pierre-Yves Schobbens (both at the University of Namur, Belgium), Necip Fazıl Ayan (Bilkent University), and the Machine Learning Group at the University of Texas at Austin, for their helpful suggestions on a preliminary version of this paper. The referees' comments have also significantly contributed to the quality 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] E. Erdem and P. Flener. Completing Open Logic Programs by Constructive Induction. Submitted.

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

[8] P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In N.E. Fuchs (ed), *Proc. of LOPSTR'97. LNCS*, Springer-Verlag, forthcoming.

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

[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] L.'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] 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.

[16] 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.

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

[18] 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.

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