A New Declarative Bias for ILP: Construction Modes

Esra Erdem[1] and Pierre Flener[2]

[1] Dept of Computer Sciences
The University of Texas at Austin
Austin, TX 78712, USA
esra@cs.utexas.edu

[2] Dept of Information Science
Uppsala University, Box 513
S–751 20 Uppsala, Sweden
Pierre.Flener@dis.uu.se

Abstract. Inductive logic programming (ILP) systems use some declarative bias to constrain the hypothesis space. We introduce a new declarative bias, called *construction modes*, capturing the required dataflow of a relation, and design a language for expressing such construction modes. Their semantics is captured via the notion of admissibility. Experiments with the ILP systems SYNAPSE and DIALOGS have established the usefulness of construction modes. Since the new bias is orthogonal to the existing search biases, it can be used in conjunction with the existing biases.

1 Introduction

In inductive logic programming (ILP) [10], a hypothesis H is to be inferred from assumed-to-be-incomplete information (or: evidence) E and background knowledge B such that $B \wedge H \models E$, where H, E, B are logic programs. Various methods are applied to constrain the hypothesis space. A relevant method is the provision of some *declarative bias*, which is any form of additional input information that restricts the hypothesis space (see [11] for a survey). One of the kinds of declarative bias that current ILP systems use is *search bias*, determining which part of the hypothesis space is searched, and how it is searched. Examples are input mode, type, and multiplicity declarations; they are combined into the single concept of mode in the ILP system PROGOL [9]. Consider, for example, the PROGOL modes for the inference of a hypothesis/program for *append*:

$$\begin{array}{l} modeh(1, append(+list, +list, -list)) \\ modeh(*, append(-list, -list, +list)) \\ modeb(1, append(+list, +list, -list)) \\ modeb(*, append(-list, -list, +list)) \\ modeb(1, +list = -integer \cdot -list) \\ modeb(1, -list = +integer \cdot +list) \end{array} \qquad (M_{append})$$

Here, *list* denotes a type. The expression $+list$ denotes a ground term of type *list*, whereas $-list$ denotes a variable of type *list*. Then, $append(+list, +list, -list)$ expresses that atoms of relation *append* have three parameters, where the first two parameters may be ground lists and the third parameter a list variable. The first line above expresses that such atoms may appear in the heads of hypothesis clauses, and that there is one correct instance of such atoms in the intended interpretation of *append*. The second line expresses that atoms of relation *append*, where the first two parameters are variable lists and the third parameter a ground list, may also appear in the heads of hypothesis clauses, and that there is an indefinite number of correct instances of such atoms. Similarly, the third and fourth lines indicate the atoms of relation *append* that may appear in the bodies of hypothesis clauses. The expression $+list = -integer \cdot -list$ says that equality ($=$) may occur in atoms of the form $L = H \cdot T$, where L is a ground list, H an integer variable, and T a list variable. The fifth and sixth lines express that equality may only appear in the bodies of hypothesis clauses, in atoms of the form $L = H \cdot T$, and that there is one correct instance if either L or both H and T are ground at call-time, with the other parameters being variables then.

Such mode declarations drastically reduce the hypothesis space, as only clauses satisfying them will be considered. However, such mode declarations are operational, as they only capture run-time properties of the atoms in clauses. Indeed, they do not capture the syntactic and semantic static relationships among the formal parameters of the relation for which a program is to be inferred.

We introduce a new search bias, called *construction modes*, that captures these static relationships in a declarative way.

For instance, consider the following program:

$$\begin{array}{l} compose(H,V,Y) \leftarrow odd(H), Y = V \\ compose(H,V,Y) \leftarrow even(H), Y = H \cdot V \end{array} \qquad (P_{compose})$$

In both clauses, the first two parameters are mandatorily used to *construct* the third parameter. This is represented by the construction mode:

$$compose(cons, cons, res)$$

where *cons* denotes the parameters used to construct the parameter denoted by res.

A construction mode does not only capture some information about the syntactic construction of the parameters, but also captures some information about their semantic construction. For instance, in the first clause of $P_{compose}$, parameter Y is equal to V provided H is odd. Here, V participates syntactically in constructing Y, whereas H participates semantically in constructing Y. In the second clause, both H and V participate syntactically in constructing Y, whereas H also participates semantically. The information about the syntactic and semantic construction of parameters is captured by the well-defined semantics of construction modes via the concept of "admissibility."

Besides, a construction mode gives information not only about construction, but also about de-construction (or: de-structuring). Consider the relation $intersection$, where $intersection(X, Y, Z)$ iff list Z is the intersection of lists X

and Y. Here the first two parameters of $intersection(X, Y, Z)$ are de-constructed into Z. This is represented by the construction mode:

$$intersection(des, des, res)$$

where des denotes the parameters that are de-structured into the parameter denoted by res.

Note that what is meant by (de-)construction is thus not operational but static: a construction mode captures the intrinsic relationship between the parameters, independently of any run-time considerations. Informally, a construction mode will thus state which parameters are (de-)constructed from which other parameters, and also expresses whether such (de-)construction is mandatory or optional. The construction modes capture the required dataflow of each involved relation. Among other things, this is how construction modes differ from "relational clichés" of [13]. Relational cliches capture the dataflow among the atoms of a clause whereas construction modes capture the dataflow among the parameters of an atom.

It is important to note that, under some conditions, the concept of construction modes introduced in this paper follows from our earlier work presented in [2]; the concept of construction modes introduced in this paper is more general.

Organization of this Paper. This paper is organized as follows. In Section 2, we formally define our new notion of construction mode, and we design a language for expressing such construction modes. In Section 3, we define *admissibility* wrt a construction mode, which captures what it means for a definite clause to satisfy the construction modes given for the relations appearing in it. Then, in Section 4, we show how construction modes have been successfully used in ILP systems. Finally, in Section 5, we review related work, outline future work, and conclude.

Notation. In expressions (i.e., literals or terms) appearing in logic programs or specifications, symbols starting with uppercase letters designate variables, whereas all other symbols designate either functions or relations, the distinction being always clear from context. All these symbols may be subscripted with natural numbers or mathematical variables (ranging over natural numbers). Unquantified variables are assumed to be universally quantified over the entire formula in which they occur. The empty list nil is also denoted by [ ], and the non-empty list $H \cdot T$ of head H and tail(-list) T (using the binary infix type constructor $\cdot$) is also denoted by $[H|T]$. Similarly, the fixed-length list $X_1 \cdot \ldots \cdot X_n \cdot nil$ is also denoted by $[X_1, \ldots, X_n]$. When we want (or need) to group several terms into a single term, we represent this as a tuple, using angled brackets. For instance, $\langle f(a, 22), g(X)\rangle$ is a term representing the couple built of the two terms $f(a, 22)$ and $g(X)$.

2 Construction Modes: A New Declarative Bias

Informally, a construction mode for a relation states which parameters are (de-)constructed from which other parameters, and also expresses whether such (de-)construction is mandatory or optional. For instance, in *append* atoms, the third parameter is mandatorily constructed from the first two parameters. Contrary to input modes, there is no notion of different usages of a relation according to construction modes. Indeed, construction modes are a static notion, whereas input modes are an operational notion. However, a construction mode may still not be unique for a given relation, because there may be several ways of expressing its dataflow. For instance, in *reverse* atoms, the second parameter is mandatorily constructed from the first parameter, or vice-versa. After introducing further notation used, we incrementally define the notion of construction mode.

2.1 Syntactic Construction

Let us first define some notions of syntactic construction.

Definition 2.1 (Leaves and vertices of a term)
The *leaves* of a term t, denoted by $leaves(t)$, are the set of variables and functions appearing in t.
The *vertices* of a term t, denoted by $vertices(t)$, are the multi-set of variables and functions appearing in t.

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

Definition 2.2 (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 one of the roles of construction modes (defined below).

The reason why we consider functions of arity higher than 0 (rather than just constants) in leaves and vertices is that we want to achieve facts such as $f(a,b) \not\sqsubseteq g(a,b)$. 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 notions of syntactic construction are much more general than the sub-term (i.e., sub-tree) notion, and this additional generality is crucial in many cases. For instance, in $delete(d,[g,e,d],[g,e])$, 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 takes place 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 partially 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 partially evaluate $Z = X$ into the head $min(X,Y,Z)$, yielding $min(X,Y,X) \leftarrow X \leq Y$, but one then cannot further partially 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

We can now introduce our new concept of construction modes.

Definition 2.3 (Construction modes)
Let r be a relation of arity n, and the m_i $(1 \leq i \leq n)$ be non-empty subsets of the set $\{cons_1, \ldots, cons_n, des_1, \ldots, des_n, may_1, \ldots, may_n, may_*, res_1, \ldots, res_n\}$ such that:

- for every i,j in $1..n$, we have that res_j is in m_i iff there is some $k \neq i$ in $1..n$ such that $cons_j \in m_k$ or $des_j \in m_k$ or $may_j \in m_k$;
- for every j in $1..n$, there is at most one i in $1..n$ such that $res_j \in m_i$.

Then $r(m_1, \ldots, m_n)$ is a *construction mode* for relation r, and m_i is a construction mode for the ith parameter of r.

For instance, $compose(\{cons_1\}, \{cons_1\}, \{res_1\})$ is a construction mode, meant to express that the third parameter is a result parameter that must be constructed from the first two parameters. Contrast this with the PROGOL modes $modeh(1, compose(+integer, +list, -list))$ and $modeh(1, compose(-integer, -list, +list))$, to see that completely different information is conveyed. For convenience, we often drop the indexes j of may_j, $cons_j$, des_j, res_j when the construction modes for all parameters have the same index. Similarly, each singleton construction mode $\{m\}$ for a parameter will often be denoted by m. So the construction mode above can also be written as $compose(cons, cons, res)$. Also, $unionInter(\{cons_1, des_2\}, \{cons_1, des_2\}, res_1, res_2)$ is a construction mode for $unionInter(A, B, U, I)$ (which holds iff lists U and I are the union and intersection, respectively, of lists A and B), expressing that result U is constructed from A and B, whereas result I is de-constructed from A and B. Note that $r(may_1, cons_1)$ is not a construction mode, because no parameter is being designated as the result of construction from the two given parameters. Also, $r(res_1)$ is not a construction mode, because it does not indicate from what parameters the given result parameter must or may be (de-)constructed. Moreover,

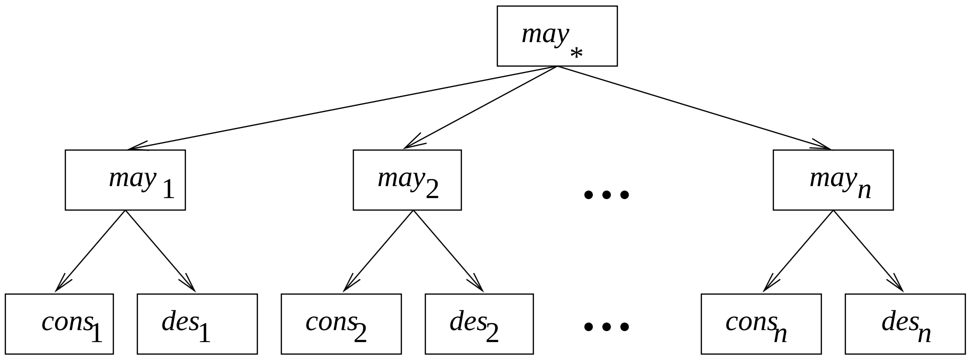

Fig. 1. Partial generality order on parameter modes

$r(cons_1, res_1, res_1)$ is not a construction mode, because a result parameter cannot possibly be in two places at the same time. Finally, $r(may_*, may_*)$ is a construction mode.

Since we do not further consider other modes in the core of this paper, we often simply speak about modes here.

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

- mode res_j means the parameter in the corresponding position is *result parameter* number j, to be (de-)constructed from other (non-result) parameters;
- mode $cons_j$ means *all* of the vertices of the parameter in the corresponding position are mandatory in syntactically constructing the parameter in the corresponding position of res_j;
- mode des_j means *some* of the vertices of the parameter in the corresponding position are mandatory in syntactically constructing the parameter in the corresponding position of res_j (hence the parameter is de-constructed into the result);
- mode may_j means the parameter in the corresponding position is optional for syntactically (de-)constructing the parameter in the corresponding position of res_j;
- mode may_* means the parameter in the corresponding position is optional for syntactically (de-)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 (in Section 3 below).

Note that mode may_* generalizes every may_j, and that each may_j itself generalizes $cons_j$ and des_j. Figure 1 illustrates this partial generality order. So the mode $reverse(cons_1, res_1)$ 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 the least general construction mode, according to the partial order of the Figure 1.

A mode may not be unique, though: for instance, $reverse(res_1, cons_1)$ is an alternative mode to $reverse(cons_1, res_1)$. However, this does not mean that there are several possible usages of the *reverse* relation according to its construction modes: contrary to the operational input modes, such as $reverse(+, -)$ or $reverse(-, +)$, which declare the possible usages of a program for a relation, construction modes are a static notion and only declare the dataflow intrinsic to the relation, independently of any usage, even if there may well be several different ways of stating it.

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

The definition of construction modes itself can be further generalized, introducing for instance a mode *ncd* (for: *neither cons nor des*), expressing that the parameter in the corresponding position may not be used for (de-)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, given enough knowledge about it.

It is important to note that the concept of construction modes introduced in this paper significantly extends the one presented in [2]. For instance, in [2], since the construction mode $must_j$ is used instead of $cons_j$ and des_j, and since the modes for the parameters of a relation can be one of $\{must_j, may_j, res_j\}$, the construction mode for the relation *unionInter* becomes $unionInter(may, may, res, res)$. Note that this does not give as useful information as $unionInter(\{cons_1, des_2\}, \{cons_1, des_2\}, res_1, res_2)$ does. It is also important to note that the concept of admissibility is extended in this paper as well. This is explained in the next section.

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. We conclude this section by discussing some properties of admissibility.

3.1 Syntactic Admissibility of an Atom wrt a Construction Mode

Let $r(m_1, \ldots, m_n)$ be a mode m for a relation r of arity n. Without loss of generality, let the indexes appearing in m run from 1 to k inclusive, where k designates the number of result parameters in m. Let $r(t_1, \ldots, t_n)$ be the considered atom. For every j in $1..k$, let $Cons_j = \langle t_i \mid cons_j \in m_i \rangle$. Similarly for May_j, May_*, Res_j. Also, for every j in $1..k$, for every i in $1..n$, let $Des_j^i = t_i$ if $des_j \in m_i$, and $Des_j^i = undefined$ otherwise.

For instance, let the construction mode be $r(may_*, \{cons_1, des_2\}, \{cons_1, des_2\}, res_1, res_2)$ and let the atom be $r(1, [a], b, [a, b], \langle\rangle)$. We then have that $n = 5$, $k = 2$, $Cons_1 = \langle [a], b \rangle$, $Cons_2 = \langle\rangle$, $Des_1^i = undefined$ for all i in $1..5$, $Des_2^2 = [a]$, $Des_2^3 = b$, $Des_2^i = undefined$ for all i in $\{1, 4, 5\}$, $May_1 = May_2 = \langle\rangle$, $May_* = \langle 1 \rangle$, $Res_1 = \langle [a, b] \rangle$, and $Res_2 = \langle\langle\rangle\rangle$.

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 $cons_j$ is mandatory in syntactically constructing the parameter in the corresponding position of res_j. Here, we should thus use syntactic containment as actual instance of syntactic construction. Formally:

$$\forall 1 \leq j \leq k \,.\, Cons_j \sqsubseteq Res_j \tag{1}$$

For instance, this is the case for the r atom and mode above. Note that $Cons_j$ groups together *all* parameters with mode $cons_j$, so that this single-iterated condition suffices, because each res_j parameter must be constructed from *all* its $cons_j$ parameters. (If the union of some sets is a subset of a given set S, then these sets are themselves subsets of S.) Also note that k may be 0, such as in $\leq (may_*, may_*)$; condition (1) then trivially holds.

Similarly, we need to express that every parameter in the corresponding position of des_j is mandatorily syntactically de-constructed into the parameter in the corresponding position of res_j. In other words, the des_j parameters, if any, are mandatorily syntactically constructed from *at least* the res_j parameter. Formally:

$$\forall 1 \leq j \leq k \,.\, \forall 1 \leq i \leq n \,.\, Des_j^i \neq undefined \rightarrow Res_j \sqsubseteq Des_j^i \tag{2}$$

For instance, this is the case for the r atom and mode above. Note that we here have to write a double-iterated condition, because each des_j parameter must *individually* be constructed from the res_j parameter. (If a set S is a subset of a union of sets, then S is not necessarily a subset of each of these sets.)

Last, we need to express that every parameter in the corresponding position of may_j is optional for syntactically (de-)constructing the parameter in the corresponding position of res_j, and that every parameter in the corresponding position of may_* is optional for syntactically (de-)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 (de-)construction. But we can refine the given approximate semantics by also requiring that the parameter in the corresponding position of res_j can *only* be syntactically constructed from the parameters in the corresponding positions of may_j, may_*, $cons_j$. Here, we should use "is syntactically obtained from" as actual instance of syntactic construction, because syntactic containment might be too strong in some cases (such as the example below). Formally:

$$\forall 1 \leq j \leq k \,.\, Res_j \subseteq \langle May_j, May_*, Cons_j \rangle \tag{3'}$$

So no leaves may be "invented" when building each Res_j. For instance, this is the case for the r atom and mode above. Note that this relationship does not hold when using syntactic containment ($\sqsubseteq$) instead of $\subseteq$.

However, this requirement is a bit too strong, as new leaves *do* sometimes appear in parameters with mode res_j. Indeed, constructors of the inductively defined type of such a parameter may appear: for instance, constant 0 and unary functor s are type constructors for Peano numbers, whereas constant nil and binary functor $\cdot$ are type constructors for lists. The atom $addPlateau(a, [\ ], [a, s(0)])$ does not satisfy condition (3') for $addPlateau(may_1, cons_1, res_1)$, because 0 and s are "invented" by the parameter with mode res_1. Since such constructors cannot really be considered new if the inductively defined type is known, we should add them to the right-hand side of (3'). Since we do not know how many times they may be "invented," we add them once and use leaf set inclusion ($\subseteq$) rather than vertex multiset inclusion ($\sqsubseteq$). Hence:

$$\forall 1 \leq j \leq k \,.\, Res_j \subseteq \langle May_j, May_*, Cons_j, nil, \cdot, 0, s, \dots \rangle \tag{3}$$

We thus here do not allow non-type-constructor leaves to be invented by res_j parameters, and leave such extensions as future work.

Hence the following overall definition:

Definition 3.1 (Atom admissibility)
An atom $r(t_1, \dots, t_n)$ is *admissible* wrt a mode m for r if conditions (1), (2), (3) are satisfied.

This concludes the refinement of a definition of syntactic atom admissibility. Let us now switch our attention to semantic clause admissibility.

3.2 Semantic Admissibility of a Clause wrt a Construction Mode

Let $r(t_1, \dots, t_n) \leftarrow \mathcal{B}$ be a definite clause, where $\mathcal{B}$ is a conjunction of atoms, called the *body* of the clause, and $r(t_1, \dots, t_n)$ 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 3.2 (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 *cons* parameters to be actually used in constructing the result parameters. Condition (1) only verifies syntactic construction, but some of the vertices of the *cons* parameters might only be used in the body $\mathcal{B}$ so as to achieve semantic construction. So condition (1) must be adapted as follows:

$$\forall 1 \leq j \leq k \, . \, Cons_j \sqsubseteq \langle Res_j, \mathcal{B}' \rangle \tag{4}$$

where $\mathcal{B}'$ is a tuple built of the atoms (seen as terms) of $\mathcal{B}$. Similarly, we want the *des* parameters to be actually de-constructed into the result parameters. Condition (2) only verifies syntactic de-construction, but some of the vertices of the *des* parameters might only be used in the body $\mathcal{B}$ so as to achieve semantic de-construction. So condition (2) must be adapted as follows:

$$\forall 1 \leq j \leq k \, . \, \forall 1 \leq i \leq n \, . \, Des_j^i \neq undefined \rightarrow Res_j \sqsubseteq \langle Des_j^i, \mathcal{B}' \rangle \tag{5}$$

Last, we want the result parameters to be constructed only from the *cons* and *may* parameters, as well as from the predefined type constructors. Condition (3) 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 (3) must be adapted as follows:

$$\forall 1 \leq j \leq k : Res_j \subseteq \langle May_j, May_*, Cons_j, \mathcal{B}', nil, \cdot, 0, s, \ldots \rangle \tag{6}$$

For instance, the head of the clause $min(X,Y,X) \leftarrow X \leq Y$ satisfies conditions (4), (5), (6) for $min(cons_1, cons_1, res_1)$, 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 (4) holds.

Conditions (4), (5), (6) can certainly be refined even further, in many different ways, but we leave this for future work. Indeed, a fine balance between the expressiveness of construction modes (i.e., the precision of the approximation of the intended relation that they achieve) and the speed of verification of admissibility has to be struck. The current definitions have evolved from a few years of experimentation and have been successfully deployed in two prototype systems (see Section 4).

For the *body atoms* now, other than their participation in conditions (4), (5), (6) above, it is necessary to verify whether they are each admissible (according to Definition 3.1) 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 3.3 (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 (4), (5), (6) 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.

It is important to note that the concept of admissibility introduced above is more general than the one presented in [2] in that it not only captures a more general definition of the construction modes but also considers the functors s and . as invented parameters besides the constants nil and 0.

3.3 Properties of Admissibility

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.

In the following, θ-subsumption [12] designates a partial generality order between clauses (by definition, a clause g *θ-subsumes* a clause s if there exists a substitution σ such that $g\sigma \subseteq s$, assuming clauses are seen as literal sets), and $lg\theta(\mathcal{C})$ denotes the least general clause, under θ-subsumption, that θ-subsumes all clauses in clause-set $\mathcal{C}$.

Lemma 3.1 (*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 a witness substitution under which c θ-subsumes d, i.e., $c\sigma \subseteq d$. 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 (4), (5), (6) 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. $\diamond$

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

Theorem 3.1 (*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 the least generalization, under θ-subsumption, of two clauses that $lg\theta(\mathcal{C})$ itself is a proper 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 3.1, 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. $\diamond$

The converse of this theorem is not true. For instance, the set

$$\begin{array}{l} insert(1,[2],[1,2]) \leftarrow \\ insert(4,[3],[3,4]) \leftarrow \end{array} \qquad (P_{insert})$$

is admissible wrt $insert(cons, cons, res)$, but its least generalization, under θ-subsumption, namely $insert(X,[Y],[K,M]) \leftarrow$, is not admissible wrt that mode.

Such properties of admissibility may be exploited in ILP systems that feature construction modes. The sample properties above could for instance be exploited if such an ILP system is based on θ-subsumption.

4 Applications of Construction Modes

We claim that construction modes may be successfully used as a declarative (search) bias in any ILP system, in addition to any other biases already used there, because of the orthogonality and thus complementarity of our new bias.

We have experimented with the usage of a simpler version of construction modes in two (related) ILP systems, namely SYNAPSE [5,3] and DIALOGS [4]. Both are schema-guided ILP systems dedicated to the inference of recursive (logic) programs, and have grown out of the tradition pioneered (in functional programming) by the THESYS system [16] and its generalization BMW_k [8]. The results of our experiments with SYNAPSE and DIALOGS, as also reported in [2], established that, with the construction modes, the resulting programs were more accurate. After explaining what "schema-guided ILP system" means, we show how construction modes can be usefully deployed on such systems.

A *hypothesis/program schema* [7] is a template program fixing the dataflow and control-flow of instance programs, plus a set of constraints (within a background theory, called the framework) on how the placeholders of the template can be instantiated. For instance, among the many possible forms of logic programs, there are the *divide-and-conquer programs* with one recursive call. They work as follows: if a distinguished formal parameter, called the *induction parameter*, say X, has a minimal value, then one can directly solve for the corresponding other formal parameter, called the *result parameter*, say Y; otherwise, X is decomposed into a smaller value T (according to a well-founded order $\prec$) by splitting off a quantity H, a sub-result V corresponding to T is computed by a recursive call, and an overall result Y is composed from H and V. Formally, this *problem-independent* dataflow and control-flow can be captured in the following template, or open program, for r:

$$\begin{array}{l} r(X,Y) \leftarrow minimal(X), solve(X,Y) \\ r(X,Y) \leftarrow \neg minimal(X), decompose(X,H,T), r(T,V), compose(H,V,Y) \end{array} \qquad (dc)$$

The place-holders, or open relations, are *minimal*, *solve*, *decompose*, and *compose*. The involved relations have the following formal specifications:

$$\begin{array}{rl}
i_r(X) \rightarrow (\ r(X,Y) \leftrightarrow o_r(X,Y)\ ) & (S_r) \\
i_r(X) \rightarrow (\ minimal(X) \leftrightarrow \neg i_{dec}(X)\ ) & (S_{min}) \\
i_r(X) \wedge \neg i_{dec}(X) \rightarrow (\ solve(X,Y) \leftrightarrow o_r(X,Y)\ ) & (S_{solve}) \\
i_{dec}(X) \rightarrow (\ decompose(X,H,T) \leftrightarrow o_{dec}(X,H,T)\ ) & (S_{dec}) \\
o_{dec}(X,H,T) \wedge o_r(T,V) \rightarrow (\ compose(H,V,Y) \leftrightarrow o_r(X,Y)\ ) & (S_{comp})
\end{array}$$

where the newly introduced symbols i_r, o_r, i_{dec}, o_{dec} must satisfy the following constraints:

$$\begin{array}{cl}
i_{dec}(X) \rightarrow \exists H,T\ .\ o_{dec}(X,H,T) & (C_1) \\
i_{dec}(X) \wedge o_{dec}(X,H,T) \rightarrow i_r(T) \wedge T \prec X & (C_2) \\
well\_founded\_order(\prec) & (C_3)
\end{array}$$

Specification S_r exhibits i_r and o_r as the input and output conditions of r, while specification S_{dec} exhibits i_{dec} and o_{dec} as the input and output conditions of *decompose*. Note that the input and output conditions of the remaining open relations are only expressed in terms of i_r, i_{dec}, o_r, and o_{dec}. The three constraints restrict *decompose* to succeed at least once if its input condition (on X) holds, and then to yield a value T that satisfies the input condition of r (so that a recursive call to r is legal) and that is smaller than X according to $\prec$, which must be a well-founded relation (so that recursion terminates). Program *dc* is correct wrt specification S_r (subject to the other specifications), within the (here omitted) framework.

Now, a closed program for *delOdds*, where $delOdds(L,R)$ holds iff R is integer-list L without its odd elements, is an instance of the schema above under the substitution

$$\begin{array}{rl}
minimal(X) \leftarrow X = [\ ] & \\
solve(X,Y) \leftarrow Y = [\ ] & \\
decompose(X,H,T) \leftarrow X = [Hd|T], H = [Hd] & \qquad (\theta) \\
compose(H,V,Y) \leftarrow odd(H), Y = V & \\
compose(H,V,Y) \leftarrow even(H), Y = [H|V] &
\end{array}$$

This substitution captures the *problem-dependent* computations of a *delOdds* program.

Schema-guided ILP systems, such as the ones mentioned above, use a hypothesis/program schema as declarative (language) bias [11]. The schema is often restricted to its template, with the specifications, constraints, and framework being omitted thus. Such systems are often dedicated to the inference of recursive programs, and even have some hardwired divide-and-conquer schema (a notable exception being DIALOGS, which is parameterized on schemas). An up-to-date overview of ILP systems, whether schema-guided or not, that are dedicated to the inference of recursive logic programs is in [6], together with a comparison on this task with selected general-purpose ILP systems. Schema-guided ILP systems dedicated to the inference of recursive programs are instances of the following (informal) program template, which infers a program P_r for relation r given evidence E_r for it:

1. *Schema-biased creation* of an open recursive program O_r that has two clauses for r, namely a non-recursive one for a base case and a recursive one for a step case. The recursive clause for r refers to an open relation q that is supposed to combine the partial results (stemming from the recursive calls) into the overall results.
2. *Abductive generation* of evidence E_q for q by running the open program O_r on evidence E_r.
3. *Inductive generalization* of the abduced positive evidence E_q^+ and analysis of the resulting closed program P_q for q: if acceptable, exit; otherwise, conjecture necessary predicate invention [14] and recursively invoke the system on the abduced evidence E_q, yielding another closed program P_q for q. In either case, the (final) program P_q is then added to open program O_r in order to get a closed program P_r for r.

The place-holders here are the schema-biased open program creation, the abductive evidence generation, the inductive evidence generalization, and the acceptability test.

For instance, when the template is *dc*, then the role of q is usually played by *compose* (or *decompose*, by duality). For instance, consider the following evidence for *delOdds*:

$$\begin{array}{rl} delOdds([\,],[\,]) \leftarrow & \\ delOdds([1],[\,]) \leftarrow & \\ delOdds([2],[2]) \leftarrow & \\ delOdds([3,4],[4]) \leftarrow & \\ delOdds([6,7,8],[6,8]) \leftarrow & \\ & \leftarrow delOdds([5],[5]) \end{array} \qquad (E_{delOdds})$$

Suppose Step 1 creates the following open program, whose (only) open relation is *compose*:

$$\begin{array}{rl} delOdds(X,Y) & \leftarrow minimal(X), solve(X,Y) \\ delOdds(X,Y) & \leftarrow \neg minimal(X), decompose(X,H,T), delOdds(T,V), compose(H,V,Y) \\ minimal(X) & \leftarrow X = [\,] \\ solve(X,Y) & \leftarrow Y = [\,] \\ decompose(X,H,T) & \leftarrow X = [Hd|T], H = [Hd] \end{array}$$

$$(O_{delOdds})$$

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

$$\begin{array}{rl} compose(1,[\,],[\,]) & \leftarrow odd(1) \\ compose(2,[\,],[2]) & \leftarrow even(2) \\ compose(3,[4],[4]) & \leftarrow odd(3) \\ compose(6,[8],[6,8]) & \leftarrow even(6) \\ & \leftarrow compose(5,[\,],[5]) \end{array} \qquad (E_{compose})$$

Suppose Step 3 induces the least generalization under θ-subsumption of the positive evidence: the result, namely $compose(H, V, Y) \leftarrow$, is not acceptable, in the sense that the overall result Y is not constructed from H and the partial result V. In other words, it is over-general. However, recursive invocation of the system on all the abduced evidence will *not* eventually yield a final program for *delOdds* that is correct wrt its informal specification above. In fact, the *compose* relation should be defined as follows (which is equivalent to the version in the beginning of this paper):

$$\begin{array}{r} compose(H, Y, Y) \leftarrow odd(H) \\ compose(H, V, [H|V]) \leftarrow even(H) \end{array} \qquad (P_{compose})$$

The SYNAPSE and DIALOGS systems overcome this flaw of such ILP systems by using construction modes. Indeed, from general programming knowledge, it is possible to state *in advance* that the construction mode of *compose* is $compose(may, cons, res)$, no matter what the relation r is. Based on this insight, a more refined method for Step 3 was developed, and even enhanced in [2], called the *Program Closing Method*. Basically, the idea is to divide (not necessarily partition) the positive evidence set E_q^+ into maximal-sized cliques (or: completely connected components), considering that there is an arc between two clauses of E_q^+ whenever their least generalization under θ-subsumption is admissible (we then say they are *compatible*), and to perform the classical approach to Step 3 for each such clique. In our example, this gives two cliques, namely:

$$\begin{array}{r} compose(1, [\,], [\,]) \leftarrow odd(1) \\ compose(3, [4], [4]) \leftarrow odd(3) \end{array} \qquad (C^1_{compose})$$

and

$$\begin{array}{r} compose(2, [\,], [2]) \leftarrow even(2) \\ compose(6, [8], [6, 8]) \leftarrow even(6) \end{array} \qquad (C^2_{compose})$$

whose least generalizations under θ-subsumption indeed are the two clauses of $P_{compose}$.

Without construction modes, this result can only be achieved under other approaches to avoiding over-generality, which are usually based on the (massive) presence of negative evidence (which must thus not be covered by any candidate hypothesis). Our approach has the pleasant advantage that the user need not present that much negative evidence, and need not even present the construction mode for the open relations in hypothesis/program templates, as they can be predetermined!

5 Conclusion

We have introduced construction modes as a new search bias for ILP systems. A construction mode captures the required dataflow of a relation, by expressing which parameters are (de-)constructed from which other parameters, as well as whether such (de-)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 parameter (de-)construction. Construction modes have been successfully employed in some ILP systems.

In terms of *related work*, there are many alternative (and complementary) notions of mode. The modes in the documentation of the primitives of the PROLOG logic programming language are *input modes*, which indicate the form (*ground* or *variable*, for instance) of each parameter upon calling a primitive. They were deduced from the programs for these primitives so as to help a posteriori specify the conditions of usage of these primitives. Deville [1] has proposed that input modes be part of a priori specification information, so that they occur in the definition of correctness of a program. He also introduced output modes, which indicate the form of each parameter upon completion of a call. He combined input modes, output modes, and multiplicity information (which indicates the minimum and maximum number of correct answers to a call) into a new concept called *directionality*, which is part of specifications along with types. This idea was (partially) picked up for the MERCURY logic programming language [18], which requires type, input mode, and multiplicity declarations to be added to the clauses of a program. Such declarations allow the compiler to infer an ordering of the clauses and of the body atoms in each clause for each input mode, such that the corresponding calls terminate correctly. The modes of the ILP system PROGOL [9], already discussed in Section 1, also include type, input mode, and multiplicity declarations.

The ILP system SIERES [17] is not really schema-guided (in the sense above), but it features a technique not unlike our Program Closing Method and its conceptual apparatus. Indeed, it also computes the least generalization under θ-subsumption of evidence (which must however be unit clauses); it constructs clauses that fit argument dependency graphs (a kind of primitive schemas that prescribe the dataflow but not the control-flow, nor any specifications, constraints, or framework); and it uses input-mode declarations (but no construction modes) to guide this construction towards non-overgeneral clauses. However, there is no notion of admissibility and compatibility, and hence no possibility of division of the evidence into cliques, i.e., no inducability of multi-clausal programs for the open relations.

The ILP system INDICO [15] is not at all schema-guided. However, it features an interesting method for conjecturing the heads of possible clauses, hence providing already much of the *discriminating* information that otherwise has to be discovered together with the *characterizing* information when starting from most-general clause heads. The method first partitions (i.e., it does not divide) the evidence (which must be unit clauses) into subsets according to the functors (e.g., type constructors) appearing in it; then it computes the least generalization under θ-subsumption 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 much more specialized than, our Program Closing Method.

In terms of *future work*, we have already mentioned interesting extensions to the definitions of construction modes (see Section 2) and admissibility (see Section 3). Of course, (refinements of) our suggested construction modes and admissibility and properties thereof may be used (maybe in conjunction with PROGOL modes) by other researchers, 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 (University of Maryland, USA), and the Machine Learning Group at the University of Texas at Austin, USA, 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. Y. Deville. *Logic Programming: Systematic Program Development.* Addison-Wesley, 1990.
2. E. Erdem and P. Flener. Completing open logic programs by constructive induction. *International J. of Intelligent Systems* 14(10):995-1019, Oct. 1999.
3. P. Flener. *Logic Program Synthesis from Incomplete Information.* Kluwer, 1995.
4. P. Flener. Inductive logic program synthesis with DIALOGS. In S. Muggleton (ed), *Proc. of ILP'96*, pp. 175–198. *LNAI* 1314, Springer-Verlag, 1997.
5. P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *J. of Symbolic Computation* 15(5–6):775–805, May/June 1993.
6. P. Flener and S. Yılmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *J. of Logic Programming* 41(2-3):141-195, Nov./Dec. 1999.
7. P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalisation of correct schemas for program synthesis. To appear in *J. of Symbolic Computation*, May 2000.
8. J.-P. Jouannaud and Y. Kodratoff. Characterization of a class of functions synthesized from examples by a Summers-like method using the Boyer-Moore-Wegbreit matching technique. In *Proc. of IJCAI'79*, pp. 440–447.
9. S. Muggleton. Inverse entailment and PROGOL. *New Generation Computing* 13:245–286, 1995.
10. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. In *J. of Logic Programming* 19–20:629–679, 1994.
11. C. Nédellec *et al.* Declarative bias in inductive logic programming. In L. De Raedt (ed), *Advances in Inductive Logic Programming*, pp. 82–103. IOS Press, 1996.
12. 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.
13. G. Silverstein, and M. Pazzani. Relational clichés: Constraining constructive induction during relational learning. *Proceedings of IWML'91*, pp. 203–207. Morgan Kaufmann, 1991.
14. I. Stahl. Predicate invention in inductive logic programming: An overview. In P.B. Brazdil (ed), *Proc. of ECML'93*, pp. 313–322. *LNAI* 667, Springer-Verlag, 1993.

15. 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.
16. P.D. Summers. A methodology for LISP program construction from examples. *J. of the ACM* 24(1):161–175, Jan. 1977.
17. 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.
18. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *J. of Logic Programming* 29(1–3):17–64, Oct./Dec. 1996.