A Refinement Operator for Theories

Liviu Badea

AI Lab, National Institute for Research and Development in Informatics
8-10 Averescu Blvd., Bucharest, Romania.
e-mail: badea@ici.ro

Abstract. Most implemented ILP systems construct hypotheses clause by clause using a refinement operator *for clauses*. To avoid the problems faced by such greedy covering algorithms, more *flexible* refinement operators *for theories* are needed. In this paper we construct a syntactically monotonic, finite and solution-complete refinement operator for theories, which eliminates certain annoying redundancies (due to clause deletions), while also addressing the limitations faced by HYPER's refinement operator (which are mainly due to keeping the number of clauses constant during refinement).
We also show how to eliminate the redundancies due to the commutativity of refinement operations while preserving weak completeness as well as a limited form of flexibility. The refinement operator presented in this paper represents a first step towards constructing more efficient and *flexible* ILP systems with precise theoretical guarantees.

1 Introduction and motivation

Although the research in Inductive Logic Programming (ILP) has concentrated on both implementations (e.g. [8, 7]) and theoretical results [4] (such as correctness, completeness and complexity), there is still a significant gap between these aspects, mainly due to a poor understanding of the combinatorial aspects of the search for solutions. In this respect, correctness and completeness results are necessary but not sufficient for obtaining an *efficient* learner. On the other hand, ad-hoc search heuristics might prove effective in certain cases, but the lack of theoretical guarantees limits their applicability and the interpretation of their results.

With only a few notable exceptions (such as HYPER [3], or MPL [5]), most implemented ILP systems construct hypotheses clause by clause by employing a *refinement operator for clauses.* As shown by Bratko [3], such greedy covering algorithms face several problems such as: unnecessarily long hypotheses (with too many clauses), difficulties in handling recursion and difficulties in learning multiple predicates simultaneously. These problems are due to the fact that a good hypothesis is not necessarily assembled from locally optimal clauses. In fact, locally inferior clauses may reveal their (global) superiority only as a whole. And it is exactly this case (of mutually interacting clauses) that most implemented ILP systems do not deal with well.

A solution to this problem would be to construct hypotheses as a whole (rather than on a clause by clause basis) by using a *refinement operator for entire theories.* But unfortunately, the combinatorial complexity of a refinement operator *for clauses* is high enough already, making a naive refinement operator *for theories* useless for all practical purposes.

The main problem encountered when constructing a refinement operator for theories is its redundancy, which heavily multiplies the size of an already huge search space. Sometimes, a good search heuristic can compensate for the size of such search spaces. However, very often, the failure in coping with the required search is attributed solely to the weakness (or maybe myopia) of the heuristic employed. In [2] we have argued that among the responsible factors for such failures one should also count the lack of *flexibility* of the refinement operator, its *redundancy*, as well as its *incompleteness*. While completeness and non-redundancy are desiderata that have been achieved in state-of-the art systems like Progol [7], flexibility has hardly been studied or even defined in a precise manner. (A precise definition of *flexibility* of refinement operators *for clauses* was given in [2].)

Flexibility becomes an issue especially in the case of (weakly) complete and non-redundant refinement operators, because redundancy is usually avoided by imposing a strict discipline on refinement operations, which usually relies on a predetermined (static) ordering of the literals, variables and even clauses. The resulting lack of flexibility can unfortunately disallow certain refinements, even in cases in which the search heuristic recommends their immediate exploration (These hypotheses will be explored eventually, but maybe with an exponential time delay.) The solution to this problem, proposed in [2], consists in enhancing the *flexibility* of the clausal refinement operator by using a dynamic literal ordering, constructed at search time. In this paper, we show how to construct a flexible refinement operator *for theories.*

Combining (weak) completeness and non-redundancy with flexibility has been studied in [2], but only for *clausal* refinement operators. Although *maximal* flexibility can only be achieved at the expense of intractability and exponential storage space, a limited form of flexibility can be achieved without significant additional costs, while preserving the completeness and non-redundancy of the refinement operator. This hints at a very general trade-off between (weak) completeness, non-redundancy, flexibility and tractability.[1]

[1] If we insist on (weak) completeness and non-redundancy, there is a fundamental trade-off between *flexibility* and *tractability*. For achieving non-redundancy, we have to store somehow a representation of the visited hypotheses space, so that every time a refinement of some H_2 to some H' is considered, we can check that H' hasn't been visited before. For tractability (of these checks), we cannot store a very fine grained representation of the visited space, so whenever visiting a hypothesis H we will store a coarse grained representation $\tilde{H}$ of H. However, this will block (in the future) not only the refinements leading to H, but also all those $H' \in \tilde{H}$ that are indiscernible w.r.t. the coarse graining. This diminishes the flexibility of the refinement operator.

A natural question is "why don't we use the partial refinement tree as a sort of index structure for the visited hypotheses space?" Although the depth of the

At the level of clauses, FOIL [8], for example, gives up completeness for maximum flexibility. But if the heuristic fails to guide the search to a solution, the system cannot rely on a complete refinement operator to explore alternative paths. On the other hand, Progol [7] insists on completeness and non-redundancy at the expense of flexibility: some refinement steps are never considered because of the static discipline for eliminating redundancies. Finally, systems based on *ideal* refinement operators are complete and can be maximally flexible, but they are highly redundant.

At the level of theories, both FOIL and Progol construct clauses one by one (Progol does this for ensuring non-redundancy and weak completeness). MPL [5] has more flexibility, but is incomplete. HYPER is less incomplete and still performs surprisingly well, even if the search heuristic were not perfect. This is mainly due to its avoiding an overly complex refinement operator such as[2]

$$\rho_T(T) = \{(T \setminus \{C\}) \cup \rho(C) \mid C \in T\}$$

by keeping the number of clauses constant during refinement:

$$\rho_H(T) = \{(T \setminus \{C\}) \cup \{C'\} \mid C \in T,\ C' \in \rho(C)\}. \tag{1}$$

Therefore, HYPER has to start with theories containing *multiple copies* of certain very general clauses (corresponding to the predicates to be learned). Thus, the main reason for HYPER's success seems to be its avoidance of certain redundancies and combinatorial explosions by keeping the number of clauses constant. But there are still other redundancies, such as:

- redundancies due to the commutativity of the refinement operations
- redundant clauses within a theory (which are not removed in order to give them later the chance to be specialized).

Keeping a constant number of clauses in theories during refinement is especially problematical when the number of clauses in the target theory cannot be easily estimated. If this number is significant, HYPER will rediscover fragments of the target theory over and over again without being able to reuse an k-clause solution fragment in a larger n-clause theory ($n > k$).[3] This also significantly increases the search time. Even worse, when learning theories for n predicates

refinement tree is typically logarithmic in the number of visited hypotheses, searching for a given hypothesis, for example a clause with literals $L_1 L_2 \ldots L_n$, involves in general searching along $n!$ paths (corresponding to all permutations of $L_1 L_2 \ldots L_n$). Of course, at most one such path will actually lead to our hypothesis (since the refinement tree belongs to a non-redundant operator), but the search along $n!$ paths *at each refinement step* cannot be avoided and is intractable in practice.

[2] Theories T are viewed as sets of clauses C. ρ is a complete refinement operator for clauses.

[3] In HYPER we also have redundancies between theories with different numbers of clauses, for example between $T_1 = T$ and $T_2 = T \wedge T_{rest}$ for a (very specific) T_{rest} such that $\forall C_2 \in T_{rest},\ \exists C_1 \in T$ with $C_1 \succeq C_2$. (This ensures that $T_1 \sim T_2$.)

$p_1, p_2, \ldots, p_n$ (while allowing at most N clauses for each of them), HYPER will have to consider N^n start theories.

In this paper we present a refinement operator for theories that solves most of the above-mentioned problems:

- it is complete and flexible (i.e. allows interleaving the refinement of clauses)
- it can exploit a good search heuristic by avoiding the pitfalls of a greedy covering algorithm
- it doesn't keep the number of clauses in a theory constant: it introduces new clauses exactly when these are needed
- it never deletes clauses (unlike MPL for example, where deleted clauses have to be marked to avoid adding them again later).

2 Refinement operators for theories

Refinement operators decouple the search heuristic from the search algorithm. Instead of the usual refinement operators for clauses, we will construct refinement operators for entire theories. For a top-down search, we deal with *downward* refinement operators, i.e. ones that construct theory *specialisations*. More precisely, we will consider refinement operators w.r.t. the *subsumption ordering* between theories.

In the following, we will regard *clauses* as sets of literals (connected by disjunction) and *theories* as sets of clauses (connected by conjunction). Clauses will be denoted by C, while theories by T (possibly with super/sub-scripts).

Definition 1. Clause C_1 *subsumes* clause C_2, $C_1 \succeq C_2$ iff there exists a substitution θ such that $C_1\theta \subseteq C_2$ (the clauses being viewed as sets of literals).

Theory T_1 *subsumes* theory T_2, $T_1 \succeq T_2$ iff $\forall C_2 \in T_2.\ \exists C_1 \in T_1$ such that $C_1 \succeq C_2$.

A hypothesis H (either a clause or a theory) *properly subsumes* H', $H \succ H'$ iff $H \succeq H'$ and $H' \not\succeq H$.

H and H' are *subsume-equivalent*, $H \sim H'$ iff $H \succeq H'$ and $H' \succeq H$.

Definition 2. A *downward refinement operator for theories* ρ_T maps theories T to sets of theories subsumed by T: $\rho_T(T) \subseteq \{T' \mid T \succeq T'\}$.

Definition 3. A refinement operator $\rho : HYP \rightarrow 2^{HYP}$ is called:

- *(locally) finite* iff $\rho(H)$ is finite and computable for all hypotheses H.
- *proper* iff for all H, $\rho(H)$ contains no $H' \sim H$.
- *complete* iff for all H and H', $H \succ H' \Rightarrow \exists H'' \in \rho^\star(H)$ such that $H'' \sim H'$.
- *weakly complete* iff $\rho^*(H_{TOP})$ covers the entire set of hypotheses HYP (H_{TOP} being the top hypothesis, for example the empty clause $\Box$ in the case of clauses, or the theory $\{\Box\}$ containing the empty clause in the case of theories).

- *solution complete* (for *theory* refinement operators only) iff for all $H \succ H'$ such that H and H' cover all positives, $\exists H'' \in \rho^\star(H)$ such that H'' covers all positives and only a subset of the negative examples covered by H'.
- *non-redundant* iff for all H_1, H_2 and H, $H \in \rho^\star(H_1) \cap \rho^\star(H_2) \Rightarrow H_1 \in \rho^\star(H_2)$ or $H_2 \in \rho^\star(H_1)$.
- *minimal* iff for all H, $\rho(H)$ contains only downward covers[4] and all its elements are incomparable ($H_1, H_2 \in \rho(H) \Rightarrow H_1 \not\succeq H_2$ and $H_2 \not\succeq H_1$).

Refinement operators have a dual nature. On the one hand, they make *syntactic* modifications to clauses and theories. On the other, these syntactic modifications have to agree with a *semantic* (generality) criterion (for a downward operator, the refinements have to be specialisations).

A refinement operator that never performs any deletions is called *syntactically monotonic* (however, such an operator may perform replacements). Syntactical monotonicity is important from a practical point of view since it avoids certain redundancies (the target of a deletion could also be reached without introducing the deleted element).

Downward refinement operators for clauses operate by adding literals and are therefore syntactically monotonic. (Adding literals to clauses produces even more specific clauses.)

However, adding clauses to theories makes these theories more general. Constructing a syntactically monotonic downward refinement operator for theories (i.e. one that doesn't delete clauses) is therefore not as simple as for clauses.

Let $\rho(C)$ be a finite and complete refinement operator for clauses. $\rho(C)$ induces the following finite and complete refinement operator for theories:

$$\begin{aligned} \rho_T(T) = \{(T \setminus \{C\}) \cup \rho(C) \mid C \in T\} \quad & \% \text{ refinement} \\ \cup \{T \setminus \{C\} \mid C \in T\} \quad & \% \text{ (clause) deletion} \end{aligned} \tag{2}$$

In other words, ρ_T either replaces a clauses $C \in T$ by (the conjunction of) *all* its refinements $\rho(C)$, or deletes a clause $C \in T$. The latter alternative (clause deletion) is necessary for completeness, although it spoils the syntactic monotonicity of ρ_T, making it highly redundant and therefore impractical.

At this point, Bratko [3] severely restricts the refinement operator to reduce its non-redundancy by keeping the number of clauses constant during refinement (see (1) above). The resulting refinement operator is however incomplete. This leaves open the question of whether a syntactically monotonic and complete refinement operator can be constructed.

3 A syntactically monotonic refinement operator for theories

In the following, we construct a *syntactically monotonic, finite and solution-complete refinement operator for theories.* (A solution-complete refinement operator may not generate all possible theories, but it will guarantee the generation

[4] H' is a *downward cover* of H iff $H \succ H'$ and no $H'' \in HYP$ satisfies $H \succ H'' \succ H'$.

of all *solutions*, i.e. theories covering all positive examples and no negative examples, while locally maximizing a certain covering heuristic.)

To start with, note that $\rho_T(T)$ replaces a clause C by *all* its clause refinements $\rho(C)$. This produces in just a few refinement steps very long theories. These could be simplified by clause deletion, but it still begs for the following question: are all clauses from $\rho(C)$ really necessary in the refinement of T?

The answer is 'no', especially when strict subsets ρ' of $\rho(C)$ are capable of covering (jointly with $T \setminus \{C\}$) all positives. Indeed, even if $(T \setminus \{C\}) \cup \rho(C)$ is in principle more general than $(T \setminus \{C\}) \cup \rho'$, the introduction of the *redundant* clauses $\rho(C) \setminus \rho'$ in the refinement of T seems unjustified, especially since it only increases theory size without improving its coverage (since $(T \setminus \{C\}) \cup \rho(C)$ is more general, it could in fact cover more negative examples! On the other hand, both theories cover all positives.[5])

For obtaining *minimal* theories using a downward operator, we should therefore only add the smallest subsets ρ' of $\rho(C)$ that preserve the covering of all positives:

$$\rho'_T(T) = \{T' = (T \setminus \{C\}) \cup \rho' \mid \text{for } C \in T \text{ and } \rho' \subseteq \rho(C) \textit{ minimal} \\ \text{(w.r.t. set inclusion) such that } T' \text{ covers } \textit{all} \text{ positives}\} \quad (3)$$

Considering a minimal[6] $\rho' \subseteq \rho(C)$ instead of the full $\rho(C)$ ensures that clauses which do not interact in (or *jointly* contribute to) covering the positives are not kept together, thereby minimizing the theory size (which is obviously important in learning).

Normally, if the clauses C_i of $\rho(C) = \{C_1, \ldots, C_n\}$ do not "interact", we re-obtain HYPER's refinement operator

$$\rho'_T(T) = \{(T \setminus \{C\}) \cup \{C_i\} \mid C \in T,\ C_i \in \rho(C)\}.$$

However, in general, clauses $C_i \in \rho(C)$ do interact. These are the cases in which our new refinement operator increases the number of clauses in the theory. This is done only when the introduction of new clauses is necessary for preserving the coverage of all positives.

For example, when refining a theory T (by refining one of its clauses $C \in T$), the number of clauses will increase only if some $C_i \in \rho(C)$ is not capable of covering[7] *all* positives by itself, so that at least some other $C_j \in \rho(C)$ is needed as well (see Figure 1): $T \mapsto (T \setminus \{C\}) \cup \{C_i, C_j\}$. Obviously, replacing C by the more specific $C_i \wedge C_j$ may avoid covering some negative examples.

The unrestricted refinement operator ρ_T makes *minimal* refinement steps (whenever it doesn't perform deletions) since $(T \setminus \{C\}) \cup \rho(C)$ is more general than $(T \setminus \{C\}) \cup \rho'$ for every $\rho' \subseteq \rho(C)$.

---

[5] When refining theories using a downward operator, we can safely discard any theory not covering all positives, since downward refinements are specialisations and therefore will not be able to extend their coverage.

[6] There can be several such minimal $\rho' \subseteq \rho(C)$.

[7] together with $T \setminus \{C\}$.

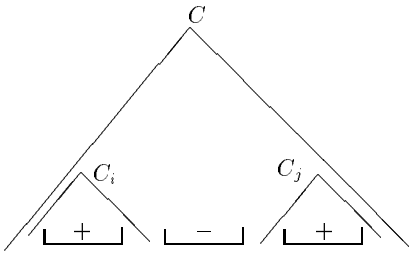

Fig. 1. C is refined to $C_i \wedge C_j$. Refining C separately to C_i or C_j would spoil the coverage of all positives, making the introduction of a new clause in the theory necessary.

However, the minimality of the refinement steps of ρ_T also involves a *significant increase in theory size*[8], which is usually not justified by the examples.[9] To make this observation more precise, we introduce a *heuristic function* for evaluating the merit of a hypothesis:

$$f(T) = pos(T) - neg(T) - |T|$$

where $|T|$ is the size of theory T, while $pos(T)$ (respectively $neg(T)$) is the number of positive (negative) examples covered by T. (f is to be maximized. Since all theories constructed by ρ'_T cover *all* positives, $pos(T) = pos$ is a constant.)

We also introduce the notion of *"compression"* realized by a theory T as

$$k(T) = pos(T) - |T|.$$

A *solution* T covers no negative examples ($neg(T) = 0$) and its size should be smaller than the number of positive examples covered: $|T| \leq pos(T)$, i.e. $k(T) \geq 0$. (Note that the compression $k(T)$ is an upper bound on the merit function $f(T) \leq k(T)$, with equality only in the case of solutions.)

Very frequently, the unrestricted ρ_T makes only very small[10] refinement steps and thus only increases the size $|T|$ without modifying the coverage $pos(T) - neg(T)$. This size increase would be justified only if the coverage would be improved. This is exactly what our improved ρ'_T does: it tolerates a size increase only if all the newly introduced clauses are necessary for covering all positives.

More precisely, let ρ' be a minimal subset of $\rho(C)$ such that $(T \setminus \{C\}) \cup \rho'$ still covers all positives. Then adding any additional (redundant) clause $C'' \in \rho(C) \setminus \rho'$ to a refinement $T' = (T \setminus \{C\}) \cup \rho'$ of T, i.e. considering $T'' = (T \setminus \{C\}) \cup \rho' \cup \{C''\}$, will not only increase the size of the resulting theory: $|T''| > |T'|$, but will also possibly increase the number of negative examples covered $neg(T'') \geq neg(T')$ (since T'' is more general than T'), thus leading to a theory that is *worse* (w.r.t. the heuristic function) than the original refinement: $f(T'') < f(T)$.

[8] which makes the resulting theories impractical in just a few refinement steps.

[9] There may be no difference in example coverage between $(T \setminus \{C\}) \cup \rho(C)$ and $(T \setminus \{C\}) \cup \rho'$. In other words, while the first theory is *intensionally* more general than the second, the two theories can be *extensionally* equivalent.

[10] small w.r.t. the generality order.

3.1 Implementing the syntactically monotonic refinement operator for theories

The pseudo-code below, implementing ρ'_T (3), avoids generating all subsets of $\rho(C)$ – it never generates supersets of the *minimal* subsets ρ'. This is realized by generating the subsets in increasing $|\rho'|$[11] and by blocking the generation of supersets of the minimal subsets covering all positives.

Subsets S' covering all positives are never added to the candidate list L, so we will never generate supersets of S' *from* S' *itself*. However, since supersets of S' could also be generated from sets that differ from S', we use *nogoods* for avoiding the generation of such supersets.[12]

Compute $\rho'_T(T)$
  $\rho'_T(T) := \emptyset$
  forall $C \in T$
    $L := [\ \emptyset\ ]$; assume $\rho(C) = \{C_1, \ldots, C_n\}$
    while L is non-empty
      extract the first $S = \{i_1, \ldots, i_k\}$ from L
      for $j = i_k + 1, \ldots, n$
        $S' = S \cup \{j\}$
        if $\nexists$ *nogood*(S'') such that $S'' \subseteq S'$ $(\star)$
          if $T' = (T \setminus \{C\}) \cup \{C_{i_1}, \ldots, C_{i_k}\}$ covers all positives
            add *nogood*(S')
            add T' to $\rho'_T(T)$ % return T'
          else append S' to L
        end if
      end for
    end while
  end for

Instead of adding T' to $\rho'_T(T)$, one could return T' (as a refinement of T) immediately and rely on backtracking to obtain alternative refinements.

For $\rho(C) = \{C_1, \ldots, C_n\}$, the above algorithm computes the subsets $\rho' \subseteq \rho(C)$ that are *minimal* w.r.t. set inclusion such that $T' = (T \setminus \{C\}) \cup \rho'$ covers *all* positives.

For example, for $\rho(C) = \{C_1, C_2, C_3\}$, if $\rho'_1 = \{C_1\}$ and $\rho'_{23} = \{C_2, C_3\}$ cover all positives[13], but $\rho'_2 = \{C_2\}$ and $\rho'_3 = \{C_3\}$ do not, we will consider only the two theory refinements corresponding to ρ'_1 and ρ'_{23}. Note that we will *not* consider the refinement $\rho'_{12} = \{C_1, C_2\}$ since it is not minimal w.r.t. set inclusion ($\{C_1, C_2\} \supseteq \{C_1\}$, which covers all positives). See Figure 2.

[11] i.e. we first generate the refinements for which ρ' is a singleton, then those where ρ' contains 2, then 3, 4, ... clauses.

[12] The nogood test $(\star)$ can be efficiently implemented (using a tree-like representation for nogood sets).

[13] together with $T \setminus \{C\}$.

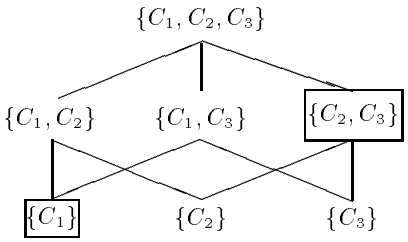

Fig. 2. Minimal subsets covering all positives.

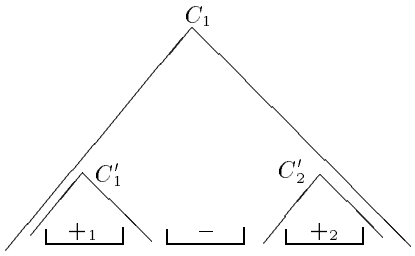

Fig. 3. C_1 is refined to C'_1 for avoiding the negatives $-$. C'_2 is introduced to cover the remaining positives $+_2$, but only when it is needed (i.e. as a refinement of C_1, and not before refining C_1).

Intuitively, considering $\{C_1, C_2\}$ (i.e. $C_1 \wedge C_2$) as a refinement would amount to considering a 2-clause theory containing C_1 even if C_1 covers by itself all positives. Now, it may be that a second clause C'_2 will be needed later to preserve the coverage of all positives (after having refined C_1 to a more specific C'_1 for avoiding negatives). However, this C'_2 need not be introduced now – it will be when needed (e.g. when refining C_1 to $C'_1 \wedge C'_2$ – see Figure 3).

Testing *all* subsets $\rho' \subseteq \rho(C)$ for minimality may pose efficiency problems due to the large number of such subsets. However, due to the syntactic monotonicity of ρ'_T, the size of theories increases during refinement, while their compression decreases monotonically. This imposes an upper bound on the size of subsets ρ' that should be considered. More precisely, when replacing clause C by some subset $\rho' \subseteq \rho(C)$ with n clauses, $|T'| = |T| - |C| + n(|C|+1)$, since the clauses $C' \in \rho'$ are obtained from C by adding a literal. For obtaining a positive compression: $0 \leq k(T') = pos - |T'| = k(T) + |C| - n(|C|+1)$,

$$n \leq \frac{k(T) - k(T') + |C|}{|C| + 1} \tag{4}$$

The upper bound (4) on the size of subsets ρ' we have to consider is not very useful in the case of high compression rates. However, we can use it in a more sophisticated implementation in which the subsets ρ' are subject to *lazy generation* (instead of being generated all at once).

More precisely, we can first construct only the refinements ρ' that guarantee a given (high) compression rate K ($k(T') = K$)[14] and then gradually decrease K until a solution is found.

Example 1. For simplicity, we consider a *propositional* example, where we can simply represent positive and negative examples for some predicate p as:
$+a, b \quad +a, c \quad -a \quad -c$

[14] using (4) to impose an upper bound on the size n of the subsets ρ'.

($+a, b$ denotes a positive example e_1, which would be represented in the usual ILP notation as $p(e_1).\quad a(e_1).\quad b(e_1).$
$+a, c$ represents $p(e_2).\quad a(e_2).\quad c(e_2).$
while $-a$ denotes the negative example $\neg p(e_3).\quad a(e_3).$)

Figure 4 depicts the associated refinement tree for the starting theory $T_0 = \{p \leftarrow\}$. The refinements of clause $p \leftarrow$ are $\rho(p \leftarrow) = \{p \leftarrow a,\ p \leftarrow b,\ p \leftarrow c\}$ and the minimal subsets covering all positives make up the theory refinements: $T_1 = \{p \leftarrow a\}$ and $T_2 = \{p \leftarrow b,\ p \leftarrow c\}$.

Then, when refining T_1 with $\rho(p \leftarrow a) = \{p \leftarrow a, b,\ p \leftarrow a, c\}$, both refinements are needed for covering all positives thus producing theory T_3 (which is a solution).

On the other hand, refining T_2 produces T_4 and T_5, only T_4 being a solution.

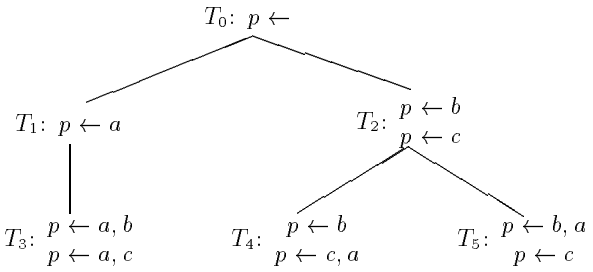

Fig. 4. A refinement tree

The search algorithm presented below uses a list of hypotheses (*Theories*), initialized with a starting theory (for example the one containing the empty clause).

solution search
  Theories := $[\{\square\}]$
  while *Theories* is nonempty
    extract T from *Theories* (according to heuristic f)
    if T is a solution **then** return T
    add $\rho'_T(T)$ to *Theories*
  end while

The refinement operator $\rho(C)$ for clauses used by ρ'_T works by adding to clause C either

- a positive literal $p(X_1, \ldots, X_n)$ (with new and distinct variables) involving a target predicate p (in the case of Horn clauses, this is allowed only if C contains no other positive literal), or
- a negative literal $\overline{p(X_1, \ldots, X_n)}$ (with new and distinct variables) involving either a target predicate or a predicate from the background theory, or

– a negative equality literal $\overline{X_i = X_j}$ involving variables X_i and X_j from C (for properness, $X_i = X_j$ should not be deducible from the equality literals of C).

4 Reducing redundancies

Although ρ'_T eliminates certain redundancies of ρ_T, other redundancies still remain. These are mainly due to the *commutativity* of the refinement operations (such as adding a literal to a clause), since all the permutations of a set of operations will now produce the same hypothesis. As shown in [1, 2], eliminating such redundancies amounts to destroying the commutativity of the refinement operations. This is equivalent to imposing a traversal discipline in the space of hypotheses and can be done by using order relations on literals, variables and clauses. As already hinted in the Introduction, a *flexible* refinement operator requires dynamic order relations (constructed at search time, rather than predetermined). [2] deals with such flexible refinement operators *for clauses*. In the following, we show how to construct flexible refinement operators *for theories* by extending the technique from [2]. Note that a straight-forward extension of the technique from [2] to theories would introduce order relations not only on variables and literals, but also on clauses. However, this would only allow constructing theories clause by clause, just like in implemented systems using refinement operators for clauses (like Progol or FOIL).

Example 2. Consider the 2 clause theory $T = \{C_1, C_2\}$ initially with $C_1 = a$, $C_2 = d$, which we want to refine first by adding literal b to C_1, then e to C_2 and finally c to C_1.

If we have *separate* (dynamic) literal and *clause* orderings, then adding b to C_1 induces the literal ordering $a < b$, then adding e to C_2 induces not only $d < e$, but also the clause ordering $C_1 < C_2$. The latter ordering will now disallow a further refinement of C_1, such as adding c to C_1. (We could have obtained the desired refinement only if all refinements of C_1, i.e. adding b and c, would have preceded the refinements of C_2. This reduces the flexibility of the refinement operator for theories and in fact we re-obtain the usual clause by clause covering approach.)

To increase the flexibility of the theory refinement operator, we shall replace the two *separate* literal and clause orderings by a single order relation on the literals from *all* clauses. Thus, instead of ordering the literals within clauses and subsequently the clauses in their entirety, we introduce a finer grained ordering between the literals of all clauses.

More precisely, the ordering will involve literal occurrences of the form $L.id(C)$ (representing literal L from clause C). Distinguishing the occurrences of the same literal in different clauses increases the flexibility of the resulting refinement operator. For example, refining $T = \{C_1, C_2\}$ with $C_1 = a$, $C_2 = b$ by adding b to C_1 and a to C_2 wouldn't be allowed if we hadn't made the above-mentioned

distinction (since an inconsistent ordering $a < b$, $b < a$ would result). With literal occurrences, we obtain the consistent ordering $a.1 < b.1$, $b.2 < b.1$, $b.1 < a.2$ (where $id(C_1) = 1$, $id(C_2) = 2$).

However, this simple approach using literal occurrences $L.id(C)$ only works whenever clauses are not "split" and therefore have a well-defined identity $id(C)$, one that does not change under refinement (as for example in HYPER). We will therefore show in the following how the redundancy of HYPER's refinement operator (1) can be reduced by adding an ordering '$<$' between literal occurrences $L.id(C)$ and one '$\prec_{id(C)}$' between variable occurrences X_i. (See [2] for a justification of the treatment of the variable ordering.)

The following refinement operator $\tilde{\rho}_H$ eliminates not only the redundancies brought about by the commutativity of the refinements of a given clause (as in [2]), but also the redundancies arising from the commutativity of refinement operations on different clauses (of the theory being refined).[15]

$T' \in \tilde{\rho}_H(T)$ iff $T' = (T \setminus \{C\}) \cup \{C'\}$ for $C \in T$, $C' \in \tilde{\rho}(C, T)$, where

$C' \in \tilde{\rho}(C, T)$ iff either

(1) $C' = C \cup \{L^{(i)}\}$ for some background literal L[16] that occurs $i-1$ times in C (i.e. $L^{(1)}, \ldots, L^{(i-1)} \in C$) and such that adding the global constraints $L^{(i)}.id(C) > T$[17] preserves the consistency of the global constraint store, $vars(C') = vars(C) \cup vars(L^{(i)})$,
(where $vars(L^{(i)})$ are the variables $\overline{X}^{(i)}$ of $L^{(i)} = p(\overline{X}^{(i)})$), or

(2) $C' = C \cup \{X_i = X_j\}$ with $X_i, X_j \in vars(C)$, such that adding the global constraints
   (a) $(X_i = X_j).id(C) > T$ and
   (b1) $X_i \prec_{id(C)} X_j$, $X_k \prec_{id(C)} X_j$ for each X_k such that $(X_i = X_k) \in C$ or $(X_k = X_i) \in C$[18],
   (b2) $X_j \prec_{id(C)} X_i$, $X_k \prec_{id(C)} X_i$ for each X_k such that $(X_j = X_k) \in C$ or $(X_k = X_j) \in C$
   preserves the consistency of the global constraint store.
   $vars(C') = vars(C) \setminus \{X_i\}$ if (b2) was applied,
   else $vars(C') = vars(C) \setminus \{X_j\}$.

---

[15] Since the *order* of clauses is important in Prolog programs, we do not attempt to eliminate the redundancies due to *permutations* of clauses (in the theory being refined), such as $C_1 \wedge C_2 = C_2 \wedge C_1$. Eliminating such redundancies would involve technical complications that are outside the scope of this paper.

[16] For a literal $L = p(\overline{X})$ with variable tuple $\overline{X}$, we introduce a standardization for the variables $\overline{X}^{(i)}$ of the i-th occurrence $L^{(i)} = p(\overline{X}^{(i)})$ of the literal L in some clause (the new and distinct variables $\overline{X}^{(i)}$ are the same for the i-th occurrence of L *on all alternative paths*).

[17] Adding $L'.id(C') > T$ to the global constraint store (L' being a literal, $id(C')$ a clause identifier and T a theory) amounts to adding $L'.id(C') > L.id(C)$ for all $C \in T$ and all $L \in C$ (or, more practically, for all *maximal* $L \in C \in T$).

[18] We add $X_i \prec_{id(C)} X_j$ only if such an $X_i = X_k$ or $X_k = X_i$ exists in C.

In both cases, (1) and (2), $id(C') = id(C)$.

$\tilde{\rho}_H$ adds either a new ordinary literal, or a new equality literal. The order relation on literals is constructed dynamically, as literals are added during successive refinements. (Of course, the consistency of the global constraint store needs to be preserved.)

Special care has to be taken for allowing multiple occurrences of a given background literal L in a clause C. For ensuring the compatibility of the induced (literal and variable) orderings on the various alternative search paths, we have to use the same variable names $\overline{X}^{(i)}$ for the i-th occurrence $L^{(i)} = p(\overline{X}^{(i)})$ of literal L *on all paths*.

Adding equalities is trickier to a certain extent due to the transitivity of equality. First, we have to avoid the trivial redundancies that would appear if we allowed adding $X_i = X_j$ for X_i and X_j already belonging to the same *cluster* of variables. (A cluster is a set of variables already unified with each other.) We do this by keeping in the set of variables candidates for unification $vars(C)$ just one representative of each variable cluster.

The constraints introduced at step *(2b)* ensure that a variable cluster $X_1 = X_2 = \ldots = X_n$ can be generated with only one sequence of refinements of type *(2)*, for example $X_1 = X_2$, followed by successively adding $X_3, X_4, \ldots, X_n$ to the growing cluster.

Example 3. For the theory $T = \{C_1, C_2\}$, the following sequence of refinements:

add literal a to C_1, add b to C_1, add c to C_2, add d to C_1, add e to C_2, add f to C_2

produces the literal ordering: $\mathbf{a.1} < \mathbf{b.1} < c.2 < \mathbf{d.1} < e.2 < f.2$, which will disallow the re-generation of the same theory by a permutation of the above operations.

Reducing the redundancies of our more general ρ'_T operator is even more complicated, mainly because of the difficulty in assigning identities to clauses obtained by "splitting". Due to space limitations, it will be the subject of a separate paper.

5 Conclusions

The refinement operator *for theories* ρ'_T presented in this paper represents a first step towards constructing more efficient and *flexible* ILP systems with precise theoretical guarantees.

Its main properties are syntactical monotonicity, solution completeness and flexibility. Flexibility allows interleaving the refinements of clauses, and thus exploiting a good search heuristic by avoiding the pitfalls of a greedy covering algorithm. On the other hand, syntactical monotonicity is important for eliminating certain annoying redundancies due to clause deletions.

We also show how to eliminate (for HYPER's refinement operator) the redundancies due to the commutativity of refinement operations while preserving a limited form of flexibility.

The paper [6] also deals with theory refinement, but the main focus is on other aspects, such as constructing bottom theories.[19] Unfortunately, the paper has several problems, which, for lack of space, cannot be discussed here.

Acknowledgments. I am grateful to Monica Stanciu for several helpful discussions on the topics of this paper.

References

1. Badea Liviu, Stanciu M. *Refinement Operators can be (weakly) perfect.* Proceedings ILP-99, pp. 21-32, LNAI 1634, Springer Verlag, 1999.
2. Badea Liviu. *Perfect Refinement Operators can be Flexible.* In Werner Horn, editor, Proceedings ECAI-2000, pp. 266–270. IOS Press, August 2000.
3. Bratko I. *Refining Complete Hypotheses in ILP.* Proceedings ILP-99, pp. 44-55, LNAI 1634, Springer Verlag, 1999.
4. Nienhuys-Cheng S.H., de Wolf R. *Foundations of Inductive Logic Programming.* LNAI 1228, Springer Verlag 1997.
5. De Raedt L., Lavrac N., Dzeroski S. *Multiple Predicate Learning.* In R. Bajcsy, editor, Proceedings IJCAI-93, pages 1037–1043. Morgan Kaufmann, 1993.
6. Midelfart H. *A Bounded Search Space of Clausal Theories.* Proc. ILP-99, 210–221.
7. Muggleton S. *Inverse entailment and Progol.* New Generation Computing Journal, 13:245-286, 1995.
8. Quinlan J.R. *Learning Logical Definitions from Relations.* Machine Learning 5:239-266, 1990.

This article was processed using the LaTeX macro package with LLNCS style

[19] While bottom *clauses* are very useful for a covering approach (since they contain the literals that *can* appear in a hypothesis), bottom *theories* are less useful, since they play the role of constraints (they specify which clauses *have to appear* in a hypothesis).