Two Advanced Transformations for Improving the Efficiency of an ILP System

Hendrik Blockeel, Bart Demoen, Gerda Janssens,
Henk Vandecasteele, Wim Van Laer
{Hendrik.Blockeel,Bart.Demoen,Gerda.Janssens,
Henk.Vandecasteele,Wim.VanLaer}@cs.kuleuven.ac.be

Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Leuven, Belgium

Abstract. Relatively simple transformations can speed up the execution of queries for data mining considerably and therefore ILP systems have since long used such optimisations. A recent publication brought to light that little is widely known on what implementations actually do. This paper describes optimizing query transformations that are implemented in existing systems as well as new transformations. Together they improve on some published simple transformations and offer a context which generalizes them. We also discuss the relationship with query pack execution and the potential use of program analysis for more powerful query transformations.

1 Introduction

Santos Costa et al. [13] describe a number of simple transformations for rendering the execution of queries for data mining more efficient. They correctly point out that little is known about how state-of-the-art ILP systems process logical queries, and what kind of efficiency considerations are made in such systems. In this paper we briefly discuss a number of feasible optimisations, some of which have been implemented in existing systems while others are ongoing work. The optimisations we discuss are related to one of the transformations proposed in [13] but also to a transformation described in [1]; they clarify the relationship between both and also include the promise of obtaining even higher efficiency gains by combining them. We furthermore discuss how these query transformation techniques can be combined with query pack execution [3], another technique for gaining efficiency.

We start with sketching the context of the transformations and a brief review of the transformations proposed in [13, 1] which are used as a starting point for our work. In Section 4 we discuss the optimisation of individual queries; this extends one of the optimisations in [13]. In Section 5 we discuss the optimisation of queries relative to other queries; this extends an optimisation described in [1]. In Section 6 we illustrate how these transformations can be combined with pack execution as described in [3]. In Section 7 we conclude.

2 The context

In the ILP setting we are interested in, a database of examples is queried repeatedly. Since the general framework is that of Horn clause logic and implementations are typically in Prolog, such a query is best seen as a conjunction of Prolog goals. Such a conjunction is repeatedly refined (i.e. goals are added) and is meant to produce eventually a part of a concept definition or predictive model. [13] describes such queries as clauses p(X) :- a(X,Y), b(Y), c(X,Z), d(Z). the function of which can be described by imagining that they are then called by a query ?- p(<X>). in which the arguments of the goal are ground. In [1], this query is described rather as the body a(X,Y), b(Y), c(X,Z), d(Z). which will be meta called but with the same variables (in this case X) ground.

Both descriptions are equivalent and in the sequel we will freely jump from one to the other.

In both descriptions, one is interested in whether the query succeeds or not and one ignores the variable bindings for which success was derived. It means that conceptually or in practice, there is a Prolog cut (!/0) after the last goal in the body. Another formulation would be to say that the body is enclosed in the meta-predicate `once/1` which is defined as `once(X) :- call(X), !.`[1]

3 Existing Work

Santos Costa et al. [13] describe a query transformation that consists of inserting cuts in the query to render its execution more efficient without changing its meaning. To this aim the query is partitioned according to the equivalence relation that is the transitive closure of the "shares body variables with" relation (where body variables are variables not occurring in the head of the clause). After grouping together literals within the same equivalence class, cuts can be inserted between the classes without influencing the semantics of the query. E.g.,

```
p(X) :- a(X,Y), b(Y), c(X,Z), d(Z).
```

is transformed into

```
p(X) :- a(X,Y), b(Y), !, c(X,Z), d(Z).
```

The justification for this transformation is the fact that p/1 is called with ground argument(s), so logically only one success is needed. In the following we will refer to this transformation as the `cut-transformation`.

Blockeel [1] describes a related method that is based on the following consideration. In TILDE [2], refining a node of the tree involves generating all refinements Q, R_i of a query Q and testing those refinements on a set of examples of which it is known that Q succeeds for them. The question that is then relevant, is: can a simpler query Q', R_i be found such that $Q \models (Q, R_i \leftrightarrow Q', R_i)$? In other words: since we already know that Q succeeds, can we use this information to

[1] The use of `once` allows a kind of nesting of cuts that is not possible when inserting cuts in the clause itself and its meta calling overhead can be transformed away with known Prolog compiler techniques.

simplify Q, R_i? Clearly, any parts of Q for which the success is independent of the success of R_i (e.g., parts of Q that are not connected to R_i via body variable chains) need not be tested again. Illustrated on the previous example, if the conjunction
`a(X,Y), b(Y), c(X,Z)`
is known to succeed for a given X, then a refinement
`a(X,Y), b(Y), c(X,Z), d(Z)`
can be tested on the example X by running
`c(X,Z), d(Z)`
instead of the full conjunction, i.e. the success of the full conjunction is equivalent to the success of the smaller one. In the TILDE system [2], a predicate called `smartcall/1` was implemented, that first simplifies a query according to the above reasoning and then meta-calls it.[2] For this reason we will refer to this optimisation as the smartcall transformation. The simplification algorithm is shown in Figure 1.

procedure SIMPLIFY(Q, *conj*) **returns** query:
  V := variables of *conj*
  repeat
    L := literals in Q that contain variables in V
    V := V $\cup$ variables in L
  until V does not change anymore
  Q' := conjunction of all literals in L
  return $\leftarrow Q'$, *conj*

Fig. 1. Simplification of queries as done by smartcall. Q is the part of the query that is known to succeed, *conj* represents the added literals.

The relationship with the cut-transformation is clear: those parts of the query that according to the cut-transformation belong to other equivalence classes than the recently added literals (i.e., those that are separated from the added literals by cuts) are actually removed by the smartcall-transformation. The cut-transformation does not assume that parts of the query are known to succeed, whereas smartcall does.

In the following two sections we show how both transformations can be extended.

4 Optimisation of individual queries

4.1 Generalisations

The cut-transformation divides a body statically in independent groups of literals and each group itself contains literals that all depend on each other. A small

[2] A similar optimisation was also implemented in the ICL system [9]. It can be controlled through the setting *simplify*.

example will suffice to show that even in such a group, one can make further optimisations: we make the assumption that head variables are ground on the call, so that for further explanation they are ignored (and in fact have been removed). Also the implicit cut at the end of the body will not be written but it is understood that we are interested in only one success.

The body:

`a(X,Y), b(Y,Z), c(Z), d(X,U), e(U)`

can be transformed (under certain conditions — see later) to

`a(X,Y), once(b(Y,Z), c(Z)), d(X,U), e(U)`

We will refer to this transformation as the `once-transformation`. We show how this can be achieved in two steps: we first present a dynamic version of this transformation and then lift it to a static transformation. We will initially assume that we want to retain the order of goals inside a group. Later we will lift this restriction as well.

4.2 Dynamic once-transformation

Consider a conjunction of goals like

`a(X,Y), b(Y,Z), c(Z), d(X,U), e(U)`

Since all goals are in the same equivalence class, we are inclined to meta call it as is. However, let us do this incrementally, i.e. execute the first goal a(X,Y) and then consider the remainder of the conjunction:

`b(Y,Z), c(Z), d(X,U), e(U)`

Assume furthermore that X and Y were ground by the call to a(X,Y). Then it is clear that the parts b(Y,Z), c(Z) and d(X,U), e(U) are no longer dependent and the whole conjunction can be replaced by

`b(Y,Z), c(Z), !, d(X,U), e(U)`

in the notation by [13], or as we prefer it:

`once((b(Y,Z), c(Z)), d(X,U), e(U)`

The whole principle can now also be recursively applied to the conjunction inside the once goal and to the remainder of the conjunction. This leads directly to a dynamic implementation of the once-transformation:

```
dyn_once(Conj) :-
        (Conj = (G,Gs) ->
            ( independent_prefix(Conj,Prefix,RestConj) ->
                once(dyn_once(Prefix)),
                dyn_once(RestConj)
            ;
                call(G),
                dyn_once(Gs)
            )
        ;
            call(Conj)
        ).
```

All the code (including auxiliary predicates) can be found in the appendix.

The above might not be the most efficient implementation, but it certainly is capable of yielding arbitrary speedups. As an artificial example, take:

```
p(X) :- a(X), b(X), !.

q(X) :- once(a(X)), b(X), !.

a(_).
a(N) :- N > 0, M is N - 1, a(M).

b(N) :- N > 0, M is N - 1, b(M).
```

`?- p(N).` has a complexity $O(N)$ whereas `?- q(N).` has complexity $O(N^2)$. The transformation of p into q is exactly what the once-transformation achieves, and its overhead is constant in N.

It remains to be seen whether big gains can be obtained in practice, i.e. in Tilde or Warmr[7]: until now, no real large conjunctions were generated for some benchmarks, exactly because the execution times were too large. Maybe the above technique will render large conjunctions feasible at last.

Note that the dynamic once-transformation can be applied unconditionally. Only when we want to lift it to a static transformation we will need extra conditions.

4.3 From dynamic to static transformation

Since the same conjunction is usually tested against many examples one clearly would like to avoid performing the dyn_once call for every example, but rather wants to perform it once on the conjunction and then call the transformed query. This is feasible when there is information on the mode of the predicates in the conjunction. In the explanation of above example, we assumed that the goal a(X,Y) grounds its arguments. Suppose that the system has this information, either by program analysis (if a/2 is a database predicate, it is obvious; but also for many background predicates, this can be derived once and for all) or by declaration by the user. Then the transformation can be done statically by the code:

```
stat_once(Conj,NewConj) :-
        copy_term(Conj,CopiedConj),
        transform(CopiedConj,NewCopiedConj),
        transform_parallel(Conj,NewCopiedConj,NewConj).

transform(Conjunction, NewConjunction) :-
        independent_prefix(Conjunction,Prefix,Rest), !,
        NewConjunction = (once(NewPrefix), NewRest),
```

```
        transform(Prefix, NewPrefix),
        transform(Rest,NewRest).
transform((Goal,Conj),NewConjunction) :-
        !,
        NewConjunction = (Goal,NewConj),
        apply_groundness_information_to(Goal),
        transform(Conj,NewConj).
transform(G,G) :- apply_groundness_information_to(G).
```

We have modeled the application of the groundness information to a goal by the predicate apply_groundness_information_to/1: one possible implementation is to ground (using numbervars/3 for instance) all the arguments that the goal is known to make ground on success. This changes the initial goal and is the reason for making the copy in the beginning and the transform_parallel goal: full code is again in the appendix.

Section 4.5 will show how also sharing information can be taken into account.

There is a relation with the work in [11, 4, 10] where automatic parallelisation of goals is attempted. One technique transforms a sequential conjunction of goals G1 and G2 into:

```
ground(G1) ->
       G1 & G2 % parallel
;
       G1 , G2  % sequential
```

[11, 4, 10] has many more transformations that could be applied also in our context.

Also [12] is interesting in this context, as it discusses transforming the Prolog cut to once/1 from a different perspective and motivation.

Note that the techniques above can be seen as a specific form of Intelligent Backtracking [5]. When introducing **once/1** we want to avoid backtracking to choicepoints not responsible for the current failure. Of course Intelligent Backtracking can have a significant runtime overhead. Our techniques should avoid most of this overhead by doing static transformations.

4.4 Changing the order in the conjunction

The above transformations respect the order of the goals in the conjunction. This is not always a good idea. Assume that the conjunction (after executing a(X,Y)) was

`b(Y,Z), d(X,U), c(Z), e(U)`

then it is clear that looking for the longest independent prefix yields no useful result. It might therefore be beneficial to change the order of the literals and obtain a better result. In the example one would benefit from interchanging the d/2 and c/1 goals before applying the dyn_once action. In general reordering of goals can have a negative impact on performance, so some care must be taken here. The reordering done in [13] makes sure that within one new component, the order is as in the original query, thus avoiding any such performance degradation. Also correctness and success need not be preserved by general change of order especially in the presence of non-logical constructs (e.g. cuts in the background knowledge) or arithmetic goals.

Code that reorders safely can be based on the dyn_once or stat_once transformation: the dynamic version is given in the appendix.

4.5 Role of program analysis in determining (in)dependent calls

The execution of a conjunction of goals can create bindings to variables in the goals and thereby influence the dependencies between the goals. It is crucial for the correctness of the optimisations that they take into account all possible dependencies. The cut-transformation does this by considering all the variables of the goals: this indeed corresponds to the (worst) case where a call creates dependencies between all its variables. The dynamic once-transformation does the same, but dynamically. The static once-transformation as defined before assumes that such dependencies do not exist and consequently can produce incorrect results: assume that in the previous example a(X,Y) binds X to Y but leaves both free, then the `stat_once` transforms incorrectly the remaining goals to (note that $X = Y$):

```
b(X,Z), c(Z), !, d(X,U), e(U)
```

It means that not only groundness but also sharing can (and must) be taken into account: in the code of `stat_once` at the place where now only groundness information is used, information about possible dependencies must be used.

This leads to the crucial questions: is there a more precise way to describe the dependencies? and how can we obtain this information? The answers can be found in the literature: in the context of automatic parallelisation of goals the **set sharing** domain [10, 4] has been proposed as a description for dependencies between variables in goals. Here we just would like to point out that set sharing can be used to refine the dependencies between calls as it also captures groundness and independence between variables. More details are found in the appendix D. Set sharing has been studied in the context of abstract interpretation for (constraint) logic programming [6]. It is a form of program analysis and aims at deriving at compile-time information about the execution of a program. The point is that abstract interpretation can be used to automatically derive the more precise information (expressed by set sharing), that we can link set sharing to the current approach, and that we can refine the optimisations based on the set sharing information.

5 Taking into account previous success

As mentioned before, when ILP systems repeatedly refine clauses, they may be able to exploit the fact that it is known that part of the clause body succeeds.

The original TILDE implementation did this by cutting the query into independent parts and dropping all but those that contain the additional literals. The reasoning is that the success of the independent parts is not influenced by the added literals. When looking at certain queries, however, the question naturally arises whether there are other situations where parts of the query can be dropped, besides independence as defined above. Consider for instance the following simple case:
`p(X) :- a(X,Y), b(Y), c(Y)`
where c(Y) is the literal added during the last refinement step. The question is under what circumstances this clause can be reduced to
`p(X) :- a(X,Y), c(Y).`
Clearly this is not always allowed: in the interpretation $\{a(1,1),\ a(1,2),\ b(1),\ c(2)\}$ the simplified query succeeds for $X = 1$ whereas the original query fails. However, the simplification is allowed if the value of Y is uniquely determined after calling `a(X,Y)`.

Situations like this occur quite frequently in ILP, because often part of the example description is essentially an attribute-value description. It is then typically the case that a predicate introduces an attribute of the example on which tests are subsequently performed. For instance, in the Mutagenesis data set molecules have an `lumo` attribute, which has one single value for each molecule. Once the attribute has been introduced, several tests may be performed; for instance, TILDE may at some point refine a query $Q =$ `lumo(X,L), L<2` by adding a test `L>0`. The resulting query $Q' =$ `lumo(X,L), L<2, L>0` is equivalent to Q'' $=$ `lumo(X,L), L>0` if it is only run on examples for which Q is known to succeed.

The smartcall optimisation algorithm can easily be adapted to take uniqueness information into account, as shown in Figure 2. The way in which the literal L should be chosen in this algorithm is not specified; best results will be obtained by choosing L so that when several literals contain the same variable but one of them uniquely determines it while the others do not, the literal that determines the variable is chosen.

procedure SIMPLIFY(Q, *conj*) **returns** query:
  V := variables of *conj*
  $Q' := \emptyset$
  repeat
    L := some literal in $Q - Q'$ that contains variables in V
    add L to Q'
    $V := V \cup$ variables in L *that are not uniquely determined*
  until no more literals L can be chosen
  return $\leftarrow Q'$, *conj*

Fig. 2. An algorithm for the extended smartcall transformation.

Note the similarity between the once-transformation and the extended smartcall: one makes use of information on groundness of variables (and hence is helped by groundness analysis) to simplify queries, the other makes use of information on uniqueness and should therefore include a kind of uniqueness analysis.

We also note that the extended smartcall brings the efficiency of ILP systems closer to that of attribute-value learners. It is well-known that the generality of ILP systems makes them less efficient even for attribute-value problems, because in general they cannot make certain assumptions that attribute-value learners do make. It would be nice if ILP systems could somehow detect cases where such simplifying assumptions can be made, and adapt their behaviour accordingly. Ideally an ILP system should behave like an attribute-value learner when given an attribute-value problem. We believe the extended smartcall is one step towards this goal.

6 Query Optimisation and Query Packs

In this section we discuss how the above described optimisations can be combined with a totally different method for improving the efficiency of ILP systems: query pack execution.

When many similar queries are run one after another, which is typically the case in ILP systems, one can expect a lot of redundancy in the computations that are performed: some of the search involved in computing success for a query q_1 will be redone when computing success for a similar query q_2. In [3] a method is described for executing a hierarchically structured set of queries (a so-called query pack) in such a way that a lot of this redundancy is removed. For instance, assume for a certain set of queries

```
grandfather(X) :- parent(X,Z), parent(Z,Y), male(X).
grandfather(X) :- parent(X,Z), parent(Z,Y), female(X).
grandfather(X) :- parent(X,Z), parent(Z,Y), male(Y).
grandfather(X) :- parent(X,Z), parent(Z,Y), female(Y).
grandfather(X) :- parent(X,Z), parent(Z,Y), male(Z).
grandfather(X) :- parent(X,Z), parent(Z,Y), female(Z).
```

it is to be decided for each single clause for which values of X it succeeds. Pack execution essentially involves running the body of the pack

```
grandfather(X) :-
                  parent(X,Z), parent(Z,Y),
                  (male(X) or female(X) or male(Y) or female(Y)
                           or male(Z) or female(Z)).
```

so that finding answer substitutions for `parent(X,Z), parent(Z,Y)` is done only once instead of 6 times. (The *or* operator is similar to Prolog's disjunction (;) but has a slightly different semantics; this difference is not important for this paper, details are in [3]). Significant speedups (up to a factor of 20) on benchmark ILP data sets are reported in [3].

6.1 Packs and Smartcall

It is possible to combine execution of query packs with the extended smartcall optimisation. This could be done by first optimising queries separately and then constructing a pack from the optimised queries. Consider, for instance, the grandfather example. The first two queries can be simplified but the last four cannot (superfluous literals are in small print):

```
grandfather(X) :- parent(X,Z), parent(Z,Y), male(X).
grandfather(X) :- parent(X,Z), parent(Z,Y), female(X).
grandfather(X) :- parent(X,Z), parent(Z,Y), male(Y).
grandfather(X) :- parent(X,Z), parent(Z,Y), female(Y).
grandfather(X) :- parent(X,Z), parent(Z,Y), male(Z).
grandfather(X) :- parent(X,Z), parent(Z,Y), female(Z).
```

The pack for the above clauses would look as follows:

```
grandfather(X) :-
                  male(X) or female(X) or
                  parent(X,Z), parent(Z,Y),
                  (male(Y) or female(Y) or male(Z) or female(Z)).
```

The difference between this pack and the previous one is easy to see when considering that X is either male or female, and hence `male(X)` or `female(X)` in the first pack can result in useless backtracking over `parent(X,Z), parent(Z, Y)` (exhausting the whole search space for answer substitutions of X, Y, Z), while in the second pack this is avoided. For large families this may make an important difference.

This procedure only works well if the part of the query that is removed ends at a branching point, otherwise one cannot remove that part of the query from the pack without duplicating another part, thus potentially losing efficiency. E.g., the pack

```
p(X) :- a(X,Y), b(Y), c(X,Z), (d(X,Y,Z) or e(X,Z)).
```

contains queries which after simplification become

```
p(X) :- a(X,Y), b(Y), c(X,Z), d(X,Y,Z).
p(X) :- a(X,Y), c(X,Z), e(X,Z).
```

from which the following pack can be constructed:

```
p(X) :- a(X,Y), (b(Y), c(X,Z), d(X,Y,Z) or c(X,Z),e(X,Z)).
```

which avoids execution of `b(Y)` in one case but only at the expense of having to execute `c(X,Z)` twice.

The point is also illustrated by Figure 3, using more graphical representations of packs. It is also clear from this figure that, at least in the example shown there, changing the order of A and B would have allowed avoiding duplication of B. In other words: up to a certain point query packs and smartcall can be combined easily, but beyond that point one has to make a choice: further transformation can be done at the expense of a less efficient pack. Reordering of literals in the member queries of the pack may in some cases push this point forward; finding out to what extent this is true is future work.

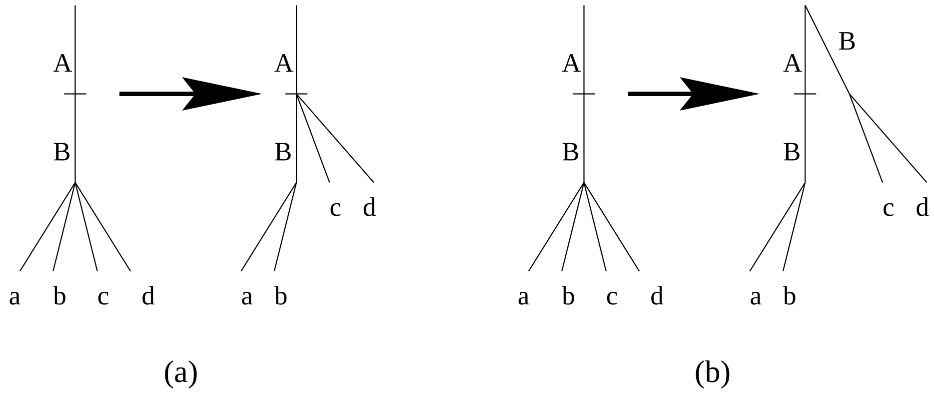

Fig. 3. Illustration of lifting branches to a node higher up in the query pack, (a) resulting in a more efficiently executable pack; (b) resulting in a pack which may be more or less efficient.

6.2 Packs and the cut- and once-transformation

The cut- and once-transformations can also be integrated to some extent into the pack execution. Consider the following pack, where `a(X,Y)` grounds Y:

```
p(X) :- a(X,Y), b(Y,Z), (c(Y,U) or d(Y,U) or c(Y,Z) or d(Y,Z))
```

The `b(Y,Z)` literal can be put inside `once` in the first 2 queries of the pack, but not in the last 2 queries. Thus, the pack can be transformed into

```
p(X) :- a(X,Y), (once(b(Y,Z)), (c(Y,U) or d(Y,U))
                 or b(Y,Z), (c(Y,Z) or d(Y,Z)))
```

which maximizes the use of the once-transformation but at the expense of duplicating the `b(Y,Z)` literal in the pack. However, this duplication is an artifact of our current pack representation; using an extended pack execution mechanism it is possible to avoid it. The idea is more or less as follows: the fact that when c(Y,U) or d(Y,U) fail, backtracking over `b(Y,Z)` should not occur, whereas when c(Y,Z) or d(Y,Z) fail it should, means that only the first success of $b(Y,Z)$ for a new value of Y must be propagated to $c(Y,U)$ and $d(Y,U)$. This can be indicated in the code as:

```
p(X) :- a(X,Y), reset(M), b(Y,Z),
                 (only_first(M), (c(Y,U) or d(Y,U)) or
                                  c(Y,Z) or d(Y,Z))
```

where the `only_first(M)` succeeds only the first time it is reached after `reset(M)` was executed. At the Prolog machine level, this is reasonably easy to implement.

7 Conclusions and future work

We have discussed two potentially powerful optimisation opportunities for ILP systems:

– **smartcall** removes superfluous literals when their success is known because of previous queries and they do not influence the success of the query; it was previously implemented in TILDE and ICL and reported on in [1]; combined with the use of groundness and uniqueness information, it generalises over these previous results
– the **once-transformation** introduces the `once` construct in a given query (possibly after having removed the superfluous part); this transformation has a dynamic and a static variant and generalises the cut-transformation proposed in [13].

It is as yet unclear to what extent the once-transformation will speed up a practical ILP system: we are currently integrating the new optimisations in our systems. We also intend to investigate the role of program analysis.

We have also indicated how the transformations can be integrated with the pack execution mechanism proposed in [3]. The pack mechanism makes it harder to apply our optimisations, in particular reordering literals in a query is an ambivalent technique because it may deteriorate the structure of a pack as well as improve it. Future work will include a study of how to reorder literals inside queries such that an optimal pack can be built from them.

We do not present practical experiments at this stage, but the fact that both in [13] and in [3] significant speedups are reported, and the fact that the techniques presented here generalise and integrate these previously published techniques, lead us to believe that even higher speedups can be achieved than those reported in either of those papers. Future work will show to what extent this is true.

Finally, one should keep in mind that while the transformations are undoubtedly useful, implementation of a good refinement operator remains an important concern of ILP system implementors. The computational overhead of a) the refinement operator, b) clause optimisation techniques and c) execution of the clauses on the data set should be so that the sum of all three should be minimised. The interaction between the three is indeed crucial as already indicated in [3] where a stronger cooperation between the refinement operator and the pack optimisation is proposed.

Acknowledgements

Hendrik Blockeel is a post-doctoral fellow of the Fund for Scientific Research (FWO) of Flanders. Henk Vandecasteele and Wim Van Laer are supported by the ALADIN Esprit Project 28.623. Presentation of this work at ILP-2000 was sponsored by the Network of Excellence in Inductive Logic Programming ILPnet2. The authors thank Ashwin Srinivasan for making available to them the text of [13], and Jan Ramon for fruitful discussions on the integration of the proposed transformations with query packs.

References

1. H. Blockeel. *Top-down induction of first order logical decision trees.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1998. `http://www.cs.kuleuven.ac.be/~ml/PS/blockeel98:phd.ps.gz`.
2. H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.
3. H. Blockeel, B. Demoen, L. Dehaspe, G. Janssens, J. Ramon, and H. Vandecasteele. Executing query packs in ILP. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference in Inductive Logic Programming*, Lecture Notes in Artificial Intelligence, London, UK, July 2000. Springer.
4. F. Bueno, M. García de la Banda, and M. Hermenegilde, Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT press, November 1994.
5. P. Codognet, and T. Sola. Extending the WAM for Intelligent Backtracking. In *Proceedings of ICLP'91, 8th International Conference on Logic Programming*, K. Furukawa(Ed.), Paris, France, MIT Press 1991.
6. P. Cousot and R. Cousot Abstract Interpretation and Application to Logic Programs *The Journal of Logic Progamming*, vol. 13, num. 2 & 3, pages 103–179, Elsevier - North-Holland, 1992.
7. L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
8. B. Demoen, G. Janssens, and H. Vandecasteele Executing query flocks for ILP. In *Proceedings of the 1999 Benelux Workshop on Logic Programming (BENELOG'99)* edited by S. Etalle, 5 november 1999, Universiteit Maastricht. Also available on the web: http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=18794
9. L. De Raedt and W. Van Laer. Inductive constraint logic. In Klaus P. Jantke, Takeshi Shinohara, and Thomas Zeugmann, editors, *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*, pages 80–94. Springer-Verlag, 1995.
10. D. Jacobs, and A. Langen, Static Analysis of Logic Programmings for Independent AND-Parallelism. *The Journal of Logic Progamming*, vol. 13, pages 291–314, Elsevier - North-Holland, 1992.
11. K. Muthukumar, F. Bueno, M. García de la Banda, M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, Vol. 38, Num. 2, pages 165-218, Elsevier - North-Holland, February 1999.
12. L. Naish, Pruning in logic programming. *Tech Report* 95/16, Department of Computer Science, University of Melbourne, Australia, jun, 1995.
13. V. Santos Costa, A. Srinivasan, and R. Camacho, A note on two simple transformations for improving the efficiency of an ilp system. In *Proceedings of the Tenth International Conference on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2000.

A Full Prolog code for dyn_once

```
dyn_once(Conjunction) :-
        (Conjunction = (A,B) ->
            ( independent_prefix(Conjunction,Prefix,Rest) ->
                once(dyn_once(Prefix)),
                dyn_once(Rest)
            ;
                call(A),
                dyn_once(B)
            )
        ;
            call(Conjunction)
        ).

independent_prefix(Conjunction,Prefix,Rest) :-
        app_conj(Prefix,Rest,Conjunction),
        independent(Prefix,Rest).

app_conj(G,Conj,(G,Conj)).
app_conj((G,C1),C2,(G,C)) :-
        app_conj(C1,C2,C).

independent(A,B) :-
        varlist(A,LA),
        varlist(B,LB),
        diff(LA,LB).

diff(L1,L2) :-
        numbervars(L1,1,_),
        notallfree(L2),
        !,
        fail.
diff(L1,L2).

notallfree([X|R]) :-
        (var(X) ->
            notallfree(R)
         ;
            true
         ).
```

B Full Prolog code for stat_once

```
stat_once(Conj,NewConj) :-
        copy_term(Conj,CopiedConj),
        transform(CopiedConj,NewCopiedConj),
        transform_parallel(Conj,NewCopiedConj,NewConj).

transform(Conjunction, NewConjunction) :-
        independent_prefix(Conjunction,Prefix,Rest), !,
        NewConjunction = (once(NewPrefix), NewRest),
        transform(Prefix, NewPrefix),
        transform(Rest,NewRest).
transform((Goal,Conj),NewConjunction) :-
        !,
        NewConjunction = (Goal,NewConj),
        apply_groundness_information_to(Goal),
        transform(Conj,NewConj).
transform(G,G) :- apply_groundness_information_to(G).
```

```
transform_parallel(Conj,NewCopiedConj,NewConj) :-
        transform_parallel1(Conj,_RestConj,NewCopiedConj,NewConj).

transform_parallel1(Conj,RestConj,(CA, CB),NewConj) :-
        !,
        transform_parallel1(Conj, RConj, CA, NewA),
        transform_parallel1(RConj, RestConj, CB, NewB),
        NewConj = (NewA, NewB).
transform_parallel1(Conj,RestConj, CA, NewA) :-
        ( CA = once(OA) ->
            transform_parallel1(Conj, RestConj, OA, NA),
            NewA = once(NA)
        ;
          Conj = (NewA, RestConj) ->
            true
        ;
            Conj = NewA   %end of Conjunction
        ).
```

C Full Prolog code for dyn_once with reordering

In the code for dyn_once/1, the call to independent_prefix/3 should be replaced by a call to independent_part/3 which can be defined by:

```
independent_part(Conjunction,Part1,Part2) :-
        conj_split(Conjunction,Part1,Part2),
        same_first_goal(Conjunction,Part1),
        independent(Part1,Part2).

conj_split((G,R),A,B) :-
         conj_split(R,X,Y),
         (
            A = (G,X), B = Y
         ;
            A = X, B = (G,Y)
         ).
conj_split((A,B),A,B).
conj_split((A,B),B,A).

same_first_goal(X,Y) :-
        (X = (A,_) -> true ; X = A),
        (Y = (B,_) -> true ; Y = B),
        A == B.
```

D Program analysis and set sharing for determining (in)dependent calls

Program analysis aims at deriving at compile-time information about the execution of a program. Typically this information could also be given explicitly by the programmer, e.g. a declaration stating that the execution of a predicate grounds it arguments. In (constraint) logic programming one has been using abstract interpretation [6] as general framework for program analysis: the concrete execution of a program is mimicked by using descriptions of the concrete substitutions, so-called *abstractions*. Much research effort has been put into the development of these abstractions, which have to be *safe* approximations of their abstract counterparts, but which also have to be *precise* enough to capture the properties of interest. Again groundness is a very well studied property. Also note that abstract interpretation computes these abstractions at all program points in a program, typically before and after each call such that the abstractions describe all possible concrete substitutions before and after a call.

For this paper we would like to point out the link with the **set sharing** abstraction [10, 4] which has been used to identify possibilities for independent AND-parallelism and which can also be used in this context to refine the dependencies between the arguments of the calls. It is beyond the scope of this paper to give a detailed explanation of the program analysis topics, but we will show in an informal way how program analysis can help in this context.

Taking into account all variables of the calls for the detection of dependency is a safe abstraction of the execution of the called predicate: execution of the predicate can possibly create dependencies between all the variables in the calls. This approach is compatible with the set sharing abstraction for a call for which no additional information is available and in which all variables are initially free independent variables – the latter is known as a goal-independent analysis. For the call p(X,Y,Z) the set sharing abstraction is the powerset of the set of all its variables {X,Y,Z}, namely {{X}, {Y}, {Z}, {X,Y}, {X,Z}, {Y,Z}, {X,Y,Z}}. A subset in the abstraction describes the possibility that the variables in the subset have in their values variables in common. The above abstraction then describes for example the concrete substitution $\{X \leftarrow f(A,B), Y \leftarrow f(A,C), Z \leftarrow g(B,A)\}$. This concrete substitution is not described by the abstraction {{X}, {Y}, {Z}, {X,Y}, {X,Z}, {Y,Z}} as the sharing of A by {X,Y,Z} is not allowed, nor by the abstraction {{X}, {Y}, {Z}} as now no shared variables are allowed. The power set safely expresses that we have to assume all possible cases, as we do not know anything about what the call actually does to its free arguments. Reasoning with the largest set of dependent variables is safe as it describes the dependencies we are interested in.

Program analysis allows us to refine the dependency relation between the arguments of calls. A first case is the grounding of variables: as soon as a variable becomes ground it should no longer be considered for dependency determination. In the set sharing abstraction the ground variables are removed from the powerset. This is compatible with the treatment of ground variables. The point here is that program analysis derives this grounding behaviour by computing a more precise abstraction for grounding calls. If the call p(X,Y,Z) grounds all its variables, the abstraction is {}. If only X is grounded, the abstraction is {{Y}, { Z}, {Y,Z }} (which still allows all possible dependencies between Y and Z). Finally, if X and Y are both grounded, the abstraction becomes {{Z}}.

A further refinement is taking into account the (in)dependence of variables appearing in the same call which can be computed by means of the set sharing abstraction. This is what is needed in the stat_once approach. For a call p(X,Y,Z) that only can create a dependency between Y and Z, the abstraction is $\{\ \{X\},\ \{Y\},\ \{Z\},\ \{Y,\ Z\}\}$. Again, taking all variables of the call would be an overestimation: one can make a distinction between the calls depending on $\{X\}$ and the ones depending on $\{Y,\ Z\}$. Alternatively, one could for determining dependent calls view p(X,Y,Z) as two independent calls p1(X) and p2(Y,Z) (or similarly as the conjunction p(X,_,_), p(_,Y,Z) such that each p/3 call can be part of another set of dependent calls).

Our program analysis based approach could be organised along the following lines. First for all predicates (describing the examples and the background knowledge) a goal-independent set sharing analysis is done which can be used to approximate the dependency property of a call (also taking into account the independence of variables). Also for builtins, set sharing can compute an abstraction. Typically, builtins such as comparison impose that the variables are ground, so the set abstraction will be empty. For X = Y, the abstraction is $\{\{X,Y\}\}$. Note this has to be done only once.
Then we consider the calls in a query from left to right together with their dependency approximation (as derived from set sharing abstractions). We will also propagate the groundness information from left to right which will remove ground variables from the dependency approximations.[3] During this left-to-right traversal we can determine the chains of dependent calls by computing transitive closures on the "grounded" dependency approximations.

Let us consider the following example query where `t(T)` is the extension.
`p(X,Y), q(X,Z), lof(Z), r(Y,T), t(T).`

1. In the most general setting, all the calls have as dependency approximation their complete set of variables. Thus they all belong to the same (dependency) chain.
2. Suppose p(X,Y) grounds both X and Y, then we have the following dependency approximation after propagating the groundness: p(X,Y):$\{\}$, q(X,Z): $\{Z\}$, lof(Z): $\{Z\}$, r(Y,T): $\{T\}$, t(T): $\{T\}$. Thus after the call p(X,Y) there are two chains: `q(X,Z), lof(Z)` and `r(Y,T), t(T)`.
   Note that backtracking over values for Z nor T is not necessary as soon as a chain has succeeded once.
3. Suppose p(X,Y) does not ground X nor Y but X and Y remain independent, then the set abstraction is $\{\{X\},\ \{Y\}\}$ and we could replace p(X,Y) by the conjunction p(X,_), p(_,Y). The dependency approximation then becomes, p(X,_): $\{X\}$, p(_,Y): $\{Y\}$, q(X,Z): $\{\{X,Z\}\}$, lof(Z): $\{\{Z\}\}$, r(Y,T): $\{\{Y,T\}\}$, t(T): $\{T\}$. We can identify two chains: `p(X,_),q(X,Z), lof(Z)` and `p(_,Y),r(Y,T), t(T)`.
4. Suppose p(X,Y) grounds only one variable, let us assume X. Then we have two chains: `q(X,Z), lof(Z)` and `p(_,Y),r(Y,T), t(T)`.

[3] In abstract interpretation terms, a goal-dependent analysis could be performed based on the goal-independent analysis results.