On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs

Manuel Hermenegildo
Francesca Rossi[1]
MCC
3500 West Balcones Center Dr.
Austin, TX 78759
herme@mcc.com, rossi@mcc.com

Abstract

This paper presents and proves some fundamental results for independent and-parallelism (IAP). First, the paper treats the issues of correctness and efficiency: after defining strict and non-strict goal independence, it is proved that if strictly independent goals are executed in parallel the solutions obtained are the same as those produced by standard sequential execution. It is also shown that, in the absence of failure, the parallel proof procedure doesn't generate any additional work (with respect to standard SLD-resolution) while the actual execution time is reduced. The same results hold even if non-strictly independent goals are executed in parallel, provided a trivial rewriting of such goals is performed. In addition, and most importantly, treats the issue of compile-time generation of IAP by proposing conditions, to be written at compile-time, to efficiently check strict and non-strict goal independence at run-time and proving the sufficiency of such conditions. It is also shown how simpler conditions can be constructed if some information regarding the binding context of the goals to be executed in parallel is available to the compiler trough either local or program-level analysis. These results therefore provide a formal basis for the automatic compile-time generation of IAP. As a corollary of such results, the paper also proves that negative goals are always non-strictly independent, and that goals which share a first occurrence of an existential variable are never independent.

1 Introduction

There has been significant interest (e.g. see [8], [11], [13], [6], [3], [5], [10], [14], [22], [23], etc.) in parallel execution models for logic programs which exploit "independent and-parallelism" (IAP). These models appear to have (in common with OR-parallel models – see [20] and its references) the very desirable characteristics of offering performance improvements through the use of parallelism, while at the same time preserving the conventional "don't

[1]On leave from the Computer Science Dept. of the University of Pisa, Italy, and supported by a grant from the Italian National Research Council.

know" semantics of logic programs. However, so far these claims remained to be formally proved: there was a need for a formal definition of goal independence and of a parallel proof procedure. Also, some results regarding the complexity of such procedure and the proof of its equivalence with the conventional one were missing.

Furthermore, goal independence can be trivially checked at run-time, but it involves significant overhead. If the objective is the automatic generation of parallelism (and/or to check user annotations for correctness), it is important to be able, at compile-time, to either identify unconditional goal independence, or construct correct and sufficient conditions for efficient run-time checking of independence. This has to be done under realistic assumptions about the binding information available to (or obtainable by) the compiler. Although some independence conditions have been informally proposed they generated redundant work, they applied only to a special case of independence ("strict independence"), and they remained to be proved sufficient. This paper attempts to fill the above mentioned gaps.

The rest of the paper proceeds as follows: in Section 2 strict goal independence and and-parallel resolution of strictly independent goals are first defined, and then it is proved that if strictly independent goals are executed in parallel the solutions obtained are the same as those produced by standard SLD-resolution. It is also proved that, in the absence of failure, the parallel proof procedure doesn't produce any additional work while the total execution time is reduced. In Section 2.4 and 2.6 two sets of efficient conditions for checking strict goal independence are proposed and proved correct, corresponding to the cases in which the goals to be run in parallel are considered in isolation or, respectively, as part of a program. Finally, non-strict independence is defined in Section 3 as a relaxation of the concept of strict independence. The corresponding properties are then proved and conditions shown for the case of parallel execution of non-strictly independent goals. The special and important cases of clauses which have existential variables and negative goals are treated respectively in Sections 2.7 and 3.3.

We also show in a running example how these results can be applied to the automatic compile-time generation of &-Prolog's Conditional Graph Expressions (CGEs) for controlling independent/restricted and-parallelism. The results presented in this paper can also be used to prove the correctness of the bit-vector method of Lin and Kumar [14], the SDDA approach of Chang et al. [5], and of the Conditional Dependency Graphs and EGE generation rules of Jacobs and Langen [12].

2 Strict Goal Independence and Parallelism

In this section we introduce the concept of strict independence of a set of goals. Then we prove that if such a set is strictly independent and the goals in the set are executed in parallel, the answer substitution obtained is identical to that which would be obtained if they were executed sequentially. Intuitively, this result guarantees that the answers to a query obtained by executing a program in independent and-parallel will be the same as those obtained from a sequential system.

Furthermore, we show that, except in the case of failure, the total amount

of work (resolution steps) performed by the parallel system is the same as in a sequential system, and we also show that the total execution time is lower than the sequential time. Finally, we show how to obtain a sufficient condition for the strict independence of a set of goals. This is done first by considering the goals in isolation, and then by considering them as part of a program. These results are of particular importance because they can be used for the generation at compile-time of sufficient conditions for strict independence which can be checked at run-time with low overhead. Thus, they represent the theoretical basis for automatic parallelization tools.

2.1 Strict Goal Independence

In the following definitions, as throughout the paper, the notation used follows that of Lloyd [15] and Apt [2, 1].

Definition 1 (variables in a goal) *: Given any goal g, let us call $var(g)$ the set of all the variables occurring in g.*■

Definition 2 (strict goal independence) *: Two goals g_1 and g_2 are said to be strictly independent for a given substitution θ iff $var(g_1\theta) \cap var(g_2\theta) = \Phi$. N goals $g_1, \ldots, g_n$ are said to be strictly independent for a given θ if they are pairwise strictly independent for θ. Also, two goals are said to be strictly independent if they are strictly independent for any θ (and again, we can extend this definition to a set of goals).*■

This same definition of strict independence can be applied also to terms without any change. Note that the above definition considers the goals after applying the substitution θ to them. This means that g_1 and g_2 may have no variables in common but at the same time they may not be strictly independent for a given θ.

Example 1 *: Let us consider the two goals $p(x)$ and $q(y)$. They do not have any variable in common, but they may be not strictly independent for some substitution. For example, given $\theta = \{x/y\}$, we have $p(x)\theta = p(y)$ and $q(y)\theta = q(y)$, so $p(x)$ and $q(y)$ are not strictly independent for this substitution, because $p(x)\theta$ and $q(y)\theta$ share the variable y. On the contrary, given $\theta = \{x/w, y/v\}$, we have $p(x)\theta = p(w)$ and $q(y)\theta = q(v)$, so $p(x)$ and $q(y)$ are strictly independent for the given θ because $p(w)$ and $q(v)$ do not share any variable. Finally, note that $p(a)$ and $q(b)$, where a and b are constants, are strictly independent for any θ.*■

Note that if a term (or a goal) is ground, then it is strictly independent from any other term (or goal). In fact, if a term is ground, its set of variables is empty, so the intersection of this set with any other set of variables will be empty, which is a sufficient condition for strict independence.

Also, note that strict independence is not transitive, i.e. if we know that g_1 and g_2 are strictly independent and also that g_2 and g_3 are strictly independent, then we cannot conclude that g_1 and g_3 are strictly independent. Let us consider for example the goals $p(x)$, $q(y)$, and $r(z)$ and the substitution $\theta = \{x/f(w), y/a, z/g(w)\}$. It is easy to see that $p(x)$ and $q(y)$ are independent w.r.t. θ, as well as $q(y)$ and $r(z)$. However, $p(x)\theta = p(f(w))$ and $r(z)\theta = r(g(w))$ so these two goals are not strictly independent w.r.t. θ, because they share the variable w.

2.2 Parallel Execution of Strictly Independent Goals

The usual SLD-resolution proof procedure is sequential, in the sense that at each step it selects only one goal in the current resolvent. If we want to run some of the goals in this resolvent in parallel (because we have discovered that they are strictly independent) we have to allow the selection of more than one goal.

Suppose that we have $G_i = (g_1, \ldots, g_n)$ as the current resolvent, and that we know that g_1 and g_2 are strictly independent. The normal SLD-resolution proof procedure with left-to-right selection rule would first select g_1 and, only when the proof of g_1 has been completed, it would select and try to prove g_2. But we know that these two goals do not share variables, so any binding created by one of them will never affect the other one. This means that we can select g_1 and g_2 simultaneously, wait for the end of their proofs and, if they both succeed, apply both the substitutions that they have generated to the remaining goals $g_3, \ldots, g_n$. Note that the *standardization apart* step (ensuring that all variables are "fresh") is still performed every time a new input clause is selected during the proofs of both g_1 and g_2, and needs to ensure that all the fresh variables are distinct in both of these proofs.

We will prove that this parallelization of SLD-resolution maintains its correctness and does not change the result of the whole proof. Also, we will prove that, in the absence of failure, it does not introduce any new work with respect to the sequential version, this meaning that in the worst case it is as efficient as its sequential version.

2.3 Strict Goal Independence is Sufficient for Parallelizing

Looking at the sequential version of the SLD-resolution proof procedure in more detail, suppose that both g_1 and g_2 can be proved true, and suppose also that the corresponding proofs take k_1 steps for g_1 and k_2 steps for g_2. This means that if we select g_1 first, we end up with the new resolvent $G_{i+k_1} = (g_2\theta_1, \ldots, g_n\theta_1)$. On the contrary, if g_2 is selected first, then we have $G_{i+k_2} = (g_1\theta_2, g_3\theta_2, \ldots, g_n\theta_2)$. We will prove that θ_1 does not affect g_2, and also that θ_2 does not affect g_1.

Theorem 1 *: If g_1 and g_2 are strictly independent and θ_i is the substitution obtained from the proof of g_i, i=1,2, then $g_2\theta_1 \equiv g_2$ and $g_1\theta_2 \equiv g_1$.*

Proof: We only prove that $g_2\theta_1 \equiv g_2$, the other result having the same proof. Due to the fact that θ_1 is obtained in k_1 steps, we have $\theta_1 = \theta_{11}, \ldots, \theta_{1k_1}$. Now, each θ_{1i} contains variables which come from $g_1\theta_{11} \ldots \theta_{1i-1}$ and/or variables from an input clause, which are all fresh variables because of the standardization apart step that is performed every time an input clause is selected. This means that $dom(\theta_{1i}) \cap var(g_2) = \Phi$ for each $i = 1, \ldots, k_1$, so also $dom(\theta_1) \cap var(g_2) = \Phi$ because $dom(\theta_1) \subseteq (var(\theta_{11}) \cup \ldots \cup var(\theta_{1k_1}))$. Thus, θ_1 cannot affect any variable in g_2, and therefore $g_2\theta_1 \equiv g_2$.■

Let us now continue to follow the two alternatives, that one in which g_1 is selected first and the other one in which g_2 is. In the first case, and using the result of Theorem 1, we have $G_{i+k_1} = (g_2, g_3\theta_1, \ldots, g_n\theta_1)$, so now

g_2 is selected and, because of the fact that no binding has been applied to it, its proof will take exactly k_2 steps (as if it were selected first). So we have $G_{i+k_1+k_2} = (g_3\theta_1\theta_2, \ldots, g_n\theta_1\theta_2)$. In the other case (first we select g_2 and then g_1) we have $G_{i+k_2+k_1} = (g_3\theta_2\theta_1, \ldots, g_n\theta_2\theta_1)$.

From the switching lemma in [15] we have that $G_{i+k_1+k_2} = G_{i+k_2+k_1}$, i.e that the order in which g_1 and g_2 are selected in the sequential SLD-resolution is not significant for the resulting binding over the remaining goals. This is true in general (also for non independent goals), but in our case we have more: because of Theorem 1 we know that g_1 and g_2 do not need to exchange any binding information, so not only can we run them sequentially in any order, but also we can run them in parallel without changing the result and without incrementing the number of resolution steps.

Thus, if we define $\theta = \theta_1\theta_2 = \theta_2\theta_1$, then after having run g_1 and g_2 in parallel we have the new resolvent $(g_3\theta, \ldots, g_n\theta)$ in $max\{k_1, k_2\}$ steps, given that two theorem provers are applied in parallel. Also, the total amount of work performed is the same as in the sequential case, i.e. $k_1 + k_2$ steps.[2]

2.4 A Correct Local Condition for Strict Goal Independence

Checking the strict independence of a set of goals in the resolvent at run-time is trivial, since it is sufficient to apply the definition. However, computing the set of variables for each goal and checking that their intersection is empty could originate large amounts of run-time overhead. In general, given a collection of goals we would like to be able to generate at compile-time an efficient, sufficient condition for their strict independence. We will first treat the case in which goals to be run in parallel are considered in isolation, rather than as part of a program. This means that we have to give a condition that does not depend on any particular substitution, but which when checked later at run time during the proof of some given query (in which those goals are called) only succeeds if the goals are strictly independent. We would of course also like that this condition involve less run-time overhead than the trivial check for goal independence. One particularly useful way of defining such a condition is as follows:

Definition 3 (*i_cond*) : *Given a collection of goals* $g_1, \ldots, g_n$, *an i_cond (independence condition) is either "true" or a conjunction of one or more of the following goals: ground(x), indep(x,y).*■

To understand the semantics of *i_cond*, note that x and y can be variables, terms, or goals, that $ground(x)$ succeeds when x is ground, and that $indep(x, y)$ succeeds when x and y do not share variables, i.e. $indep(x, y)$ corresponds to a test for *goal-* and/or *term independence* as defined in Section 2.1.
Also, for syntactic convenience, we can write $indep([(x_1, y_1), \ldots, (x_m, y_m)])$ which is equivalent to $indep(x_1, y_1), \ldots, indep(x_m, y_m)$ and we can also write $indep([(x_1, [y_{11}, \ldots, y_{1n}]]), \ldots$ instead of the equivalent form $indep(x_1, y_{11})$, $\ldots$, $indep(x_1, y_{1n}), \ldots$.

[2] In a practical implementation this is only true, of course, if the parallelism overhead is sufficiently low. However, low overhead appears to be attainable in most cases as demonstrated by systems such as RAP-WAM [10, 9] and APEX [14].

Example 2 *:*

- *ground*(x) *fails for the substitutions* $\theta_1 = \{x/f(y)\}$, $\theta_2 = \{x/y\}$, *and* $\theta_3 = \{x/f(g(1,y,3))\}$ *but succeeds for all of* $\theta_1 = \{x/f(a)\}$, $\theta_2 = \{x/a\}$, *and* $\theta_3 = \{x/f(g(1,2,3))\}$;
- *indep*(x,y) *fails for the substitutions* $\theta_1 = \{x/y\}$, $\theta_2 = \{x/f(w), y/[1,2,w]\}$, *and* $\theta_3 = \{x/f(g(1,y,3))\}$ *but succeeds for all of* $\theta_1 = \{x/f(w), y/[1,2,w]\}$, $\theta_2 = \{x/f(a)\}$, *and* $\theta_3 = \{x/f(y), y/a\}$;
- *ground*$([a,b,c])$ *always succeeds, while indep*$(f(x),g(y))$ *succeeds for* $\theta = \{x/a\}$, *but not for* $\theta = \{y/x\}$.■

For an *i_cond* to be locally correct w.r.t. strict independence, we have to be sure that such *i_cond* is sufficient for the strict independence of the goals $g_1, \ldots, g_n$:

Definition 4 (local correctness of an *i_cond* w.r.t. strict ind.) *: An i_cond is said to be locally correct w.r.t. strict independence for a set of goals* $g_1, \ldots, g_n$ *iff, for any substitution* θ, *it holds that if i_cond* θ *succeeds then* $g_1, \ldots, g_n$ *are strictly independent for* θ.■

As mentioned before, a trivial (but very inefficient) locally correct *i_cond* w.r.t. strict independence can always be constructed as the conjunction of all $indep(goal_i, goal_j) \forall i,j, i \neq j$. We will now show how to generate at compile time a more efficient, locally correct *i_cond* w.r.t. strict independence for a set of goals.

Definition 5 (SVG,SVI) *: Given a collection of goals* $g_1, \ldots, g_n$, *let us define the two sets SVG and SVI as follows:*

- $SVG = \{v$ *such that* $\exists i,j, i \neq j$ *with* $v \in var(g_i)$ *and* $v \in var(g_j)\}$;
- $SVI = \{(v,w)$ *such that* $v \notin SVG$ *and* $w \notin SVG$ *and* $\exists i,j, i \neq j$ *with* $v \in var(g_i)$ *and* $w \in var(g_j)\}$.■

Let us now consider a particular *i_cond*, that is

$$ground(SVG), indep(SVI).$$

We will show that this *i_cond* is locally correct w.r.t. strict independence.

Theorem 2 *: The i_cond* $ground(SVG), indep(SVI)$, *where SVG and SVI are computed on the collection of goals* $g_1, \ldots, g_n$, *is locally correct w.r.t. strict independence for those goals.*

Proof: We will prove the theorem for n = 2. The extension to a larger number of goals is straightforward and based on the same idea. We have to prove that, for any substitution θ, if both $ground(SVG\theta)$ and $indep(SVI\theta)$ succeeds, then $var(g_1\theta) \cap var(g_2\theta) = \Phi$. Recall that $ground(SVG\theta)$ succeeds iff $\forall v \in SVG$, $v\theta$ is ground, and $indep(SVI\theta)$ succeeds iff $\forall (v_1, v_2) \in SVI$, $var(v_1\theta) \cap var(v_2\theta) = \Phi$. Now, suppose, by contradiction with what we want to prove, that there exists a variable v in $var(g_1\theta) \cap var(g_2\theta)$. This variable v can be the result of applying θ either to some variable already shared by g_1 and g_2, or to two different variables occurring in g_1 and g_2. In any case, a contradiction arises. In fact:

- $v \neq x\theta \forall x \in SVG$, because we have assumed $ground(SVG\theta)$ to succeed, and
- $\nexists(v_1, v_2)$ with $v_1 \in var(g_1)$ and $v_2 \in var(g_2)$ such that $v \in var(v_1\theta)$ and $v \in var(v_2\theta)$ because we have assumed $indep(SVI\theta)$ to succeed.

This means that no variable can be in $var(g_1\theta) \cap var(g_2\theta)$, so g_1 and g_2 are strictly independent.■

Precise conditions for the local correctness of an *i_cond* w.r.t. strict independence were first proposed, to the best of our knowledge, in [10]. Those conditions are herein proved correct and enhanced by checking independence on a minimal set of pairs of variables (rather than on a list, which can incur in unnecessary checks). For efficiency reasons, we can improve the conditions further by grouping pairs in SVI which share a variable x, such as $(x, y_1), \ldots, (x, y_n)$ (let us call this set of pairs S_x), by writing only one pair of the form $(x, [y_1, \ldots, y_n])$, let us call it P_x. Computing $indep(x, y)$ implies, at least in a naive implementation, computing the $var(x)$ and $var(y)$ sets, and then their intersection. Although $indep(x, [y_1, \ldots, y_n])$ can be expanded syntactically to the n $indep(x, y_1), \ldots, indep(x, y_n)$ checks, it can be more efficiently implemented directly by first computing $var(x)$, and then each one of the $var(y_i)$ sets, checking their intersection with $var(x)$. In this way, the $var(x)$ set needs to be computed only once.

When, for some two variables x and y appearing in SVI, we have $S_x \cap S_y = S \neq \Phi$, then we have to choose whether the pairs in S are to be grouped in P_x or in P_y. This choice should be made in such a way that it minimizes the number of variable occurrences in the resulting SVI.

Example 3 : *The following table lists a series of sets of goals, their associated SVG and SVI sets, and a correct local i_cond w.r.t. strict independence:*

Goals	SVG	SVI	i_cond
$p(x), q(y)$	Φ	$\{(x, y)\}$	$indep(x, y)$
$p(x), q(x)$	$\{x\}$	Φ	$ground(x)$
$p(x), q(y), r(y)$	$\{y\}$	Φ	$ground(y)$
$p(x, y), q(x, y)$	$\{x, y\}$	Φ	$ground([x, y])$
$p(x, y), q(y, z)$	$\{y\}$	$\{(x, z)\}$	$ground(y), indep(x, z)$
$p(y, z), q(w)$	Φ	$\{(y, w), (z, w)\}$	$indep([(y, w), (z, w)])$
$p(y, z), q(w)$	Φ	$\{(w, [y, z])\}$	$indep([(w, [y, z])])$

■

2.5 Application Example: Local Correctness of CGEs w.r.t. Strict Independence

The results presented in the previous sections apply in general to all parallel execution models for logic programs which exploit IAP. As an example, in this section we will apply such results to a particular approach: independent/restricted and-parallelism. This approach combines compilation technology and parallel execution: it introduces parallelism in a given program by adding at compile time "graph expressions" to some clauses. The

evaluation of such expressions results in parallel execution of sets of goals at run-time. The discussion will be presented in terms of the "RAP-WAM" model [10]. This model extends DeGroot's seminal work on *Restricted AND-Parallelism* [8] by providing backward execution semantics, improved graph expressions (&-Prolog's "Conditional Graph Expressions" –CGEs– and other constructs),[3] and an efficient implementation model based on the Warren Abstract Machine (WAM) [19]. &-Prolog, the source language in this model, is basically Prolog, with the addition of the parallel conjunction operator "&" and a set of parallelism-related builtins, which includes several types of groundness and independence checks, and synchronization primitives. Parallel conditional execution graphs (which cause the execution of goals in parallel if certain conditions are met) can be constructed by combining these elements with the normal Prolog constructs, such as "->" (if-then-else). For syntactic convenience (and historical reasons) an additional construct is also provided: the Conditional Graph Expression (CGE).

Definition 6 (CGEs) *: A CGE (Conditional Graph Expression) is a structure of the form* (*i_cond* => $goal_1$ & $goal_2$ & ... & $goal_N$), *where the i_cond is an independence condition as defined previously, and each* $goal_i$, $i = 1, \ldots, n$, *is either a literal or (recursively) a CGE.* ■

CGEs appear as literals in the bodies of clauses. From an operational (Prolog) point of view, a CGE can be viewed simply as syntactic sugar for the &-Prolog clause:

(*i_cond* -> $goal_1$ & $goal_2$ & ... & $goal_N$
; $goal_1$, $goal_2$, ... , $goal_N$)

Therefore, the operational meaning of the CGE is "check *i_cond*, if it succeeds execute the $goal_i$ in parallel, else execute them sequentially." Since $goal_i$ can themselves be CGEs, CGEs can be nested in order to create more complex execution graphs.

Definition 7 (local correctness of a CGE w.r.t. strict ind.) *: A CGE* (*i_cond* => $goal_1$ & $goal_2$ & ... & $goal_N$) *is said to be locally correct w.r.t. strict independence iff i_cond is a correct local condition for* $goal_1, \ldots,$ $goal_n$ *w.r.t. strict independence.*■

I.e., a CGE is locally correct w.r.t. strict independence if for any substitution θ, it holds that if *i_cond* θ succeeds then $goal_1, \ldots, goal_n$ are strictly independent for θ.

The problem of automatically annotating a given program with &-Prolog constructs (such as CGEs), for which the results in this paper are fundamental, involves repeatedly selecting (grouping) a particular set of goals, generating a correct *i_cond* for their independence, and rewriting the program so that the selected goals are executed in parallel only if the *i_cond*

[3] &-Prolog's constructs offer Prolog syntax –so that it is possible to view the annotation process as a rewriting of the original program– and permit *conjunctions* of "checks," thus lifting limitations in the expressions proposed by DeGroot which prevented the use of the *i_cond*s presented in this paper.

succeeds. Heuristic measures can be used in the goal selection process, based on minimizing the overhead involved in the evaluation of *i_cond*, maximizing the probability of success of *i_cond*, and granularity considerations. Further discussion of these heuristics is outside the scope of this paper (see [18] for more details). A system which automatically performs such an annotation process, MA3, and which also uses global information as described in section 2.6, is described in [21, 9]. Some examples of locally correct CGEs w.r.t. strict independence are given in the following example.

Example 4 : *The following CGEs are locally correct w.r.t. strict independence:*

```
( indep(X,Y)                          => p(X) & q(Y)         ).
( ground(X)                           => p(X) & q(X)         ).
( ground([X,Y])                       => p(X,Y) & q(X,Y)     ).
( ground(Y),indep(X,Z)                => p(X,Y) & q(Y,Z)     ).
( ground(Y)                           => p(X) & q(Y) & r(Y)  ).
( ground(X),indep([[Y,W],[Z,W]]) => p(X,Y,Z) & q(X,W)   ).
```

■

2.6 A Correct Global Condition for Strict Goal Independence

In Section 2.4 we described a way to write a correct local condition for the strict independence of a given set of goals. We also proved that such a condition, obtained using only local information about the goals, is sufficient for the strict independence of the goals. However, if the goals in the set are not considered in isolation, but rather as part of a clause, sometimes this condition may be too strong, i.e. it may be that simpler *i_cond*s than those presented in Section 2.4 are sufficient for guaranteeing the strict independence of these goals for the substitutions affecting them in any proof which can be constructed with the given clause. Performing clause-level analysis in order to gather information about such substitutions is common, for example, in current Prolog compilers. Furthermore, if the whole program in which the clause appears is also considered, even more information can be available at compile-time (at least in "abstract" form) regarding the substitutions affecting these goals in any proof which can be constructed with the given clause in the given program. This is, for example, the case if global analysis techniques are applied to the program (e.g. see [5, 7, 21, 16, 4]).

In order to handle the availability of such information, we define now a new kind of correctness for an *i_cond* which is less strong than the previous one but it is still sufficient for the strict independence of the goals under such circumstances. To do that, we first have to introduce some other concepts.

Given a logic program, during the proof of some goal its variables can be bound to any term of its first order language. This set of terms can be infinite, but an elegant way to represent it in a finite structure is by using an abstract domain, i.e. a finite set each element of which is used to represent an entire class of terms. This switch from the set of all terms to the abstract domain is then used to replace substitutions with abstract substitutions that bind variables to elements of the abstract domain. Obviously, any abstract substitution represents a possibly infinite set of normal substitutions.

Example 5 : *Consider the following abstract domain* $\{free, any, ground, bottom\}$. *A possible abstract substitution binds a free variable to "free", a variable bound to a ground term to "ground", a variable that cannot be bound consistently with others to "bottom", and any other variable to "any".*■

Definition 8 (entry mode) : *An entry mode, or query form, for a given program is a query whose arguments belong to a given abstract domain.*■

Thus, an entry mode may represent a possibly infinite set of queries for the given program.

Example 6 : *The query form* $p(ground, ground)$ *represents all the possible queries* $p(t_1, t_2)$ *where* t_1 *and* t_2 *are ground terms.*■

Definition 9 (global correctness of an *i_cond* w.r.t. strict ind.) : *Let us consider a program P, a collection of goals* $g_1, \ldots, g_n$ *in the body of a clause of P and a set S of entry modes for P. Let p be a path which leads from an actual query represented in S to this collection of goals, and let* θ *be the composition of all the substitutions in p. Also, let SS be the set of all the* θ*s in all such paths. An i_cond is said to be globally correct w.r.t. the strict independence of* $g_1, \ldots, g_n$ *iff, for each* θ *in SS, it holds that if i_cond*θ *succeeds then* $g_1, \ldots, g_n$ *are strictly independent for* θ.■

In words, we relax the local correctness of an *i_cond* w.r.t. strict independence by restricting our attention from the set of all the substitutions to the set of the substitutions that can really occur at the considered point of the program. Note that the set SS is represented in the abstract domain by the l.u.b. of all the abstract substitutions corresponding to the substitutions in SS.

The following example shows that we are really relaxing the definition, because there exist some *i_cond*s that are globally correct but not locally correct for strict independence, i.e. the set of locally correct *i_cond*s is included in the set of globally correct *i_cond*s.

Example 7 : *Consider the program*
$p(x) \leftarrow q(x), r(x), s(x).$
$q(a).$
Suppose that we want to parallelize the execution of $r(x)$ *and* $s(x)$ *in the first clause. Following the approach of Section 2.4, we would consider the i_cond* $ground(x)$*, which we already know to be locally correct. Let us now consider the query form* $p(any)$*, which represents queries such as* $p(a)$ *and* $p(x)$*. The SS set for* $r(x), s(x)$ *given this query form is* $\{\{x/a\}\}$ *(and, in abstract form, perhaps* $\{\{x/ground\}\}$*), so it can be easily seen that the empty i_cond, which is not locally correct, is on the contrary globally correct.*■

As we did for local correctness, we will now show how to write, for a given set of goals, an *i_cond* that is always globally correct for the strict independence of those goals.

Definition 10 (SVG_g,SVI_g) : *Given a logic program P and a sequence of goals* $g_1, \ldots, g_n$ *appearing in the body of some clause of P, we define the two sets* SVG_g *and* SVI_g *as follows:*

- $SVG_g = SVG - \{x$ *such that* $\forall\theta \in SS\ x\theta$ *is ground.*$\}$;
- $SVI_g = SVI - \{(x,y)$ *such that* $\forall\theta \in SS\ x\theta$ *and* $y\theta$ *are strictly independent*$\}$.■

Note that the set SS used in the above definition is as in Definition 9.

Now we can consider, given a logic program P and a sequence of goals $g_1,\ldots,g_n$ appearing in the body of some clause of P, the *i_cond*

$(ground(SVG_g), indep(SVI_g))$.

It is worth noting that, while in general this *i_cond* has less checks than the corresponding locally correct one, thus gaining in efficiency at run time, it needs to use the global information about the given logic program and query form in order to construct the SVG_g and SVI_g sets.

Theorem 3 *: Given a logic program P, a sequence of goals* $g_1,\ldots,g_n$ *appearing in the body of some clause of P, and the two sets* SVG_g *and* SVI_g *as defined in Definition 10, the i_cond* $(ground(SVG_g), indep(SVI_g))$ *is globally correct for the strict independence of these goals.*

Proof: Obvious from the definition of global independence and of the sets SVG_g and SVI_g.■

Example 8 (global correctness of CGEs w.r.t. strict ind.) *: The CGE in the following clause is globally correct w.r.t. strict independence, given* $SS = \{\theta_1,\theta_2\}, \theta_1 = \{x/f(a), y/a, z/w\}, \theta_2 = \{x/b, y/b, z/a, w/b\}$ *(represented perhaps by* $\{x/ground,\ y/ground,\ z/any,\ w/any\}$*):*

```
s(X,Y,Z,W) :- ( indep(Z,W) => p(X,Y,Z) & q(X,W) ).
```

Note that $SVG = \{x\}$, $SVI = \{(y,w),(z,w)\}$, $SVG_g = \Phi$, *and* $SVI_g = \{(z,w)\}$. *Also, note that this same CGE is not locally correct w.r.t. strict independence.*■

2.7 Existential Variables

In this section we will treat the case of clauses in which existential variables (defined below) occur. This case turns out to be of great practical importance. We will show that, by looking at such variables and using the definition of global correctness of *i_cond*s, it is in some cases possible to predict the unconditional failure of an *i_cond* corresponding to a collection of goals and in others to simplify the *i_cond* by simply looking at the clause containing such goals.

Definition 11 (existential variable) *: A variable x which appears in a clause C is an existential variable iff it doesn't appear in the head of C.*■

Definition 12 (first occurrence) *: The first occurrence of a variable x in a clause C is the leftmost occurrence of that variable.*■

Theorem 4 *: Consider a collection G of goals in the body of a given clause, and the set* V_{ex} *of existential variables appearing in G. If any variable in* V_{ex} *occurs in more than one goal of G, and one of these occurrences is the first occurrence of that variable, then the goals in G are not strictly independent.*

Proof: Since an existential variable does not appear in the head of the clause, it cannot be bound before its first occurrence. Therefore, the goals cannot be strictly independent because they share a variable.■

Theorem 5 : *Consider a collection G of goals in the body of a given clause, and the set V_{ex} of existential variables appearing in G. Consider also the set F of all the variables of V_{ex} which appear only once in G and such that this one occurrence is their first occurrence. Then the variables in F are (pairwise) strictly independent.*

Proof: Consider two variables in F, say x and y. Again, since an existential variable cannot be bound before its first occurrence, $var(x) = \{x\}$ and $var(y) = \{y\}$, so $var(x) \cap var(y) = \Phi$.■

This means that the independence condition for each pair of these variables can be avoided from the *i_cond*.

Except for cases such as those which will be treated in Section 3, the appearance of existential variables in a clause implies a "hard" data dependency between goals and it can be used as the primary heuristic in the goal selection (grouping) process mentioned in Section 2.5.

Example 9 (existential variables and CGEs) : *The CGE in the following clause is globally correct w.r.t. strict independence:*
`s(X,Y) :- ( ground(Y) => p(X,Y) & q(Y,Z) ), t(Y,Z).`
Note that the indep(x,z) check is not required. However, note that this CGE is not locally correct w.r.t. strict independence. Conversely note that the following CGE, although locally correct w.r.t. strict independence, can never succeed since $p(x,y)$ and $q(x,y)$ cannot be strictly independent:
`s(X) :- ( ground([X,Y]) => p(X,Y) & q(X,Y) ).`■

3 Non-Strict Goal Independence

In order to open the possibility of making more goals available for parallel execution, in this section we relax the concept of independence, by allowing the goals to be run in parallel to share some variables, even if only in a restricted situation. We will first define a new independence condition, called non-strict independence, and then show properties of the parallel execution of non-strictly independent goals which are similar to those of strictly independent goals.

Definition 13 (v- and nv-binding) : *A binding x/t is called a v-binding if t is a variable, otherwise it is called an nv-binding.*■

Definition 14 (non-strict independence) : *Consider a collection of goals $g_1, \ldots, g_n$ and a given substitution θ. Consider also the set of shared variables $SH = \{v \mid \exists i,j, 1 \leq i,j \leq n, i \neq j, v \in (var(g_i\theta) \cap var(g_j\theta))\}$ and the set of goals containing each shared variable $G(v) = \{g_i \mid v \in var(g_i\theta), v \in SH\}$. Let θ_i be the answer substitution for $g_i\theta$. The given collection of goals is non-strictly independent for θ if the following two conditions are satisfied:*

- *for all $v \in SH$, at most the rightmost $g \in G(v)$ nv-binds v;*

- $\forall i = 1, \ldots, n$, *if* $var(g_i\theta)$ *contains more than one shared variable, say* $x_1, \ldots, x_k$, *then* $x_1\theta_i, \ldots, x_k\theta_i$ *are strictly independent.*■

Intuitively, the first condition of the above definition requires that at most one goal further instantiates a shared variable. The choice of the rightmost goal is not arbitrary. In fact, not only do we want to parallelize goals that do not need any communication during their execution, but also we would like not to introduce any additional work with respect to the sequential execution. If the goal that nv-binds x is g_i and there is another goal g_j, with $j > i$, that also contains x, then in the sequential execution with the usual left to right selection rule the execution of g_i will restrict the search space of g_j, because θ_i will affect g_j. In the parallel execution, g_j will be executed as it is (without any further instantiation of x), therefore leading to a possibly greater number of steps.

The second condition eliminates the possibility of aliases (of different shared variables) which are created by the execution of one of the parallel goals. In fact, an alias is not only a restriction of the search space (to be avoided because of the reason just discussed), but it also creates an indirect dependence among different shared variables.

A particular case of non-strict independence was hinted at by DeGroot in the "qsort" example given in [8]. The MA3 system, presented in [21], incorporated an earlier concept of non-strict independence. The two problems that the above definition tries to characterize and avoid have also been informally discussed by Winsborough and Waern in [23].

Corollary 1 *If a collection of goals is strictly independent for a given* θ, *then it is also non-strictly independent for* θ.

Proof: The two conditions in the definition of non-strict independence are always satisfied for a collection of strictly independent goals. In fact, strictly independent goals do not share any variable, while the two conditions are only about shared variables.■

3.1 Non-Strict Independence is Sufficient for Parallelizing

In this section we will show that if we have a set of goals that are non-strictly independent, then we can run them in parallel (after a variable renaming phase) while obtaining the same result as in the usual proof procedure. This is a significant improvement on the result of Section 2.3, in the sense that we now show that we can relax the condition of strict independence and still maintain parallelism. But it is now necessary to perform a global analysis of the program to check this independence condition, i.e. no locally correct *i_cond* can in general be generated for checking the non-strict independence of a set of goals. It has been shown, however, that the overhead involved in such an analysis can be reasonable ([21]).

Let us consider a sequence of goals $g_1, \ldots, g_n$ in the current resolvent G_i, and the sequential SLD-resolution as proof procedure. Suppose also that $g_j = g_j^*\theta$ for each j, that g_1^* and g_2^* are non-strictly independent for θ, that each g_j can be proved in the given program in k_j steps, for $j = 1, 2$, and that θ_j is the binding obtained by proving g_j, for $j = 1, 2$.

If g_1 is selected first, then we have $G_{i+k_1} = (g_2\theta_1, \ldots, g_n\theta_1)$. Now, it cannot be proved, as we did in Section 2.3, that $g_2\theta_1 = g_2$, because there may be some variables shared by g_1 and g_2 that have been bound in θ_1. A simple example can show that:

Example 10 *Consider the program:*
$p(z,z)$.
$q(a)$.
and the goal $\leftarrow p(x,y), q(y)$. *The two goals* $p(x,y)$ *and* $q(y)$ *are non-strictly independent, because only* q *will nv-bind the shared variable* y *and* p *contains only one shared variable. If we use a left-to-right selection rule, we have* $\theta_1 = \{x/z, y/z\}$ *from the proof of* $p(x,y)$, *and* $\theta_2' = \{z/a\}$ *from the proof of* $q(y)\theta_1$, *i.e* $q(z)$. *So at the end we obtain the substitution* $\theta = \theta_1\theta_2' = \{x/a, y/a, z/a\}$. *If on the contrary we use a right-to-left selection rule, then we have* $\theta_2 = \{y/a\}$ *from the proof of* $q(y)$, *and* $\theta_1' = \{x/a, z/a\}$ *from the execution of* $p(x,y)\theta_2$, *i.e.* $p(x,a)$. *So the final substitution in this case is again* $\theta = \theta_2\theta_1' = \{y/a, x/a, z/a\}$, *which is the same as before as we know from the switching lemma in [15]. The main thing to note here is that* $\theta_1 \neq \theta_1'$ *and* $\theta_2 \neq \theta_2'$, *which means that* $p(x,y)$ *and* $q(y)$ *need to exchange some kind of binding information during their proofs. In fact, if we try to prove them in parallel as they are, we will obtain* θ_1 *from the proof of* $p(x,y)$, θ_2 *from the proof of* $q(y)$, *and* $\theta_1\theta_2 = \{x/z, y/z\}$ *or* $\theta_2\theta_1 = \{y/a, x/z\}$ *as final substitution, that is obviously incorrect.*■

We will now show that a simple renaming of the shared variables in g_1 and g_2 will solve the problem and will allow us to run the two goals in parallel without changing the result of the whole proof and maintaining the same number of resolution steps.

In this section we talk about the parallelization of only two non-strictly independent goals, but the result can be easily generalized to an arbitrary number of goals. While there is no conceptual difference in the overall proof, the renaming algorithm for two goals is only a special case of the general one, so we will here describe the general algorithm.

Definition 15 : *Given a collection of non-strictly independent goals* $g_1, \ldots, g_n$, *let us define, for every* x *shared by two or more of the goals and nv-bound by one of them,* $X(i)$ *as the set of all the occurrences of the variable* x *in the goal* g_i. *Also, let us call* $b(x)$ *the goal that is going to nv-bind* x. *Now, for every* x, *we rename all the occurrences of* x *except those contained in* $b(x)$, *such that, for every* g_i, *all the occurrences of* x *in* $X(i)$ *have the same name and, given* g_i *and* g_j, *the occurrences of* x *in* $X(i)$ *have a name different from the name of those in* $X(j)$. *Then, for every new variable* x' *introduced in this renaming process of* x, *we add a new goal* $x' = x$ *to the given collection.*■

The idea is that we want different occurrences of x in different goals to have different names. These goals of the form $x' = x$ are called "back-binding" goals, and are somewhat related to the back-unification goals defined in [13], and the closed environment concept of [6].

Now we can show that the problem described in Example 10 can be solved by the renaming.

Example 11 *Let us consider again the program of Example 10 and the collection of goals* $p(x,y), q(y)$. *The renaming algorithm will obtain the new collection* $p(x,y'), q(y), y' = y$. *Now we execute* $p(x,y')$ *and* $q(y)$ *in parallel obtaining the respective answer substitutions* $\theta_p = \{x/z, y'/z\}$ *and* $\theta_q = \{y/a\}$. *Then we execute the goal* $y' = y$ *with the substitution* $\theta_{pq} = \theta_p\theta_q$ *applied (note that* $\theta_p\theta_q = \theta_q\theta_p$ *because the two goals are strictly independent after the renaming), that is* $z = a$, *which returns the obvious answer substitution* $\theta_b = \{z/a\}$. *Now it is easy to see that* $\theta_{pq}\theta_b = \{x/a, y'/a, y/a, z/a\}$, *which is equivalent to the original* θ *computed in Example 10. Thus we obtain the correct answer substitution.*■

Let us now show the result of the renaming algorithm on a more complicated collection of goals.

Example 12 *Consider the collection of goals* $(r(x,z,x), s(x,w,z), p(x,y), q(y))$ *in the resolvent. Suppose, in order to make them non-strictly independent, that* $p(x,y)$ *is the only goal that will nv-bind* x, $q(y)$ *is the only one nv-binding* y, *and that no goal will nv-bind* z. *According to the renaming algorithm, we will write this new collection of goals:*
$r(x',z,x'), s(x'',w,z), p(x,y'), q(y), x' = x, x'' = x, y' = y$.■

Note that in the case of only two given goals, every shared variable introduces at most one new variable, while in the general case it can introduce as many new variables as the number of goals in which it occurs.

Having dealt with the need to rename shared variables, we can now continue with the problem of showing that non-strict independence is sufficient for parallelizing.

Starting with the given collection of goals $g_1, \ldots, g_n$, where g_1 and g_2 are non-strictly independent, we will obtain, after the renaming, the new collection of goals $g_1', g_2', r_1, \ldots, r_k, g_3, \ldots, g_n$, where g_i' is g_i with some variables renamed (as described in the algorithm), and $r_1, \ldots, r_k$ are the back-binding goals introduced by the renaming.

It is easy to see that the renaming algorithm does not change the result of the proof of the given goals, since the renaming simply explicitates some bindings. So the two collections of goals $g_1, \ldots, g_n$ and g_1', g_2', $r_1, \ldots, r_k$, $g_3, \ldots, g_n$ produce the same final substitution (up to bindings of the new variables). But now, due to the renaming, g_1' and g_2' are obviously strictly independent (in fact, the renaming makes sure that they do not share any variable), so they can be run in parallel, as shown in Section 2.3.

Also, due to the first condition of the definition of non-strict independence, we are also sure not to introduce any additional step in the parallel execution of a collection of non-strictly independent goals.

Now we have to consider the steps in the execution of the back-binding goals $r_1, \ldots, r_k$. Each one of them is of the form $x' = x$, where x' is a new variable introduced by the renaming of x. While x can be nv-bound during the proof of g_2, x' is v-bound by all goals. So the goal $x' = x$ always succeeds and needs only one resolution step.

Thus, we can conclude that g_1' and g_2' can be run in parallel without changing the result of the whole proof. Also, the number of steps necessary to prove them goes from $k_1 + k_2$ to $max(k_1, k_2) + k$ if two theorem provers to be used in parallel are available.

3.2 A Correct Global Condition for Non-Strict Independence

Given a logic program P and a collection of goals $g_1, \ldots, g_n$ in the body of some clause of P, and assuming that some amount of global information about the bindings occurring in P is available at compile-time, we would like to be able to write a condition (i.e., an *i_cond*) on the variables in these goals that is sufficient to guarantee their non-strict independence at run-time, i.e., a condition similar to that of Section 2.6 but applied to non-strict independence. However, it is important to note that whereas determining strict independence only requires knowledge of θ, non-strict independence requires information on θ_i as well, which cannot be obtained from an *i_cond* check previous to the parallel execution of the goals. This information can only be obtained from global analysis and, therefore, only a *global* independence condition can be generated for non-strict independence. Therefore, we only define global correctness of an *i_cond* w.r.t. non-strict independence.

Definition 16 (global corr. of an *i_cond* w.r.t. non-strict ind.) *: An i_cond is said to be globally correct w.r.t. non-strict independence for a set of goals $g_1, \ldots, g_n$ in a program P and with a set of entry modes SS iff, $\forall \theta \in SS$, if i_condθ succeeds, then $g_1, \ldots, g_n$ are non-strictly independent for θ.*■

Above, the set SS is as defined in Definition 9. Also, in the following paragraphs SVI and SVG are as defined in Section 2.4, and SVG_g and SVI_g are as in Section 2.6.

The main difficulty in generating a globally correct *i_cond* for non-strict independence comes again from the fact that the definition of non-strict independence is given in terms of variables in θ and θ_i, whereas during compilation, and unless an extremely sophisticated global analysis is available, we can only refer to variables in the program. Therefore, we would like to translate the conditions in definition 14 into conditions involving the program variables. The nature of such conditions will of course be very closely tied to the power of the global analysis. Here we present conditions corresponding to a type of information which appears feasible to obtain with current abstract interpretation techniques: information about whether program variables will be v- or nv-bound at run-time, and about the possible sharing of variables among terms. Relatively conventional abstract analyzers can obtain the former kind of information. Recently, such techniques have been extended in order to accurately obtain the latter kind of information, as in [17]. Given such global information, a set S can be constructed which contains all shared program variables which are known to be v-bound in all θs in SS, whose occurrences are all v-bound by all g_i in which they appear, except at most the rightmost one, and which are independent from other variables in S appearing in the same goal. It will later be shown that the variables in S meet the conditions in the definition of non-strict independence.

Given the set S, consider the set $SD = S \times S - \{(x,x), x \in S\} - \{(x,y), \exists g_i, x \in g_i, y \in g_i\}$ (i.e., the set of pairs of variables of S that need to be checked for independence), the set of non-shared variables $SI = \{x$ such that x appears in at least one pair in $SVI_g\}$, and the set SSI of pairs

of variables in S and SI which may be dependent, i.e., $SSI = \{(x, y)$ such that $x \in S$ and $y \in SI$ and they are in different goals$\}$.

Definition 17 (SVG_{ns}, SVI_{ns}) *: Given a logic program P and a sequence of goals $g_1, \ldots, g_n$ in the body of some clause of P, we define two sets SVG_{ns} and SVI_{ns} as follows:*

- $SVG_{ns} = SVG_g - S$;
- $SVI_{ns} = (SVI_g \cup SD \cup SSI) - SIP$,
  where SIP is the set of pairs in $(SVI_g \cup SD \cup SSI)$ which are known to be independent due to global analysis.■

In words, SVG_{ns} contains all SVG_g except those variables meeting the non-strict independence conditions (S). SVI_{ns} makes sure that, in addition to the normal pairs to be checked for strict independence (SVI_g), also variables in S are mutually independent and independent from those in the pairs in SVI_g. The pairs that are known to be already independent (SIP) are, of course, excluded.

Now we can consider this particular *i_cond*:

$$ground(SVG_{ns}), indep(SVI_{ns}).$$

The following theorem shows that this *i_cond* is sufficient for the non-strict independence of $g_1, \ldots, g_n$.

Theorem 6 *: The i_cond $(ground(SVG_{ns}), indep(SVI_{ns}))$, where SVG_{ns} and SVI_{ns} are computed on the collection of goals $g_1, \ldots, g_n$ and by using abstract analysis on the given program P, is globally correct w.r.t. non-strict independence for those goals.*

Proof: The definition of non-strict independence imposes conditions on the set of variables actually shared by the goals $g_1, \ldots, g_n$. These variables will appear in one or more of the terms to which the variables in the program are bound by the θs in SS. Such program variables belong in principle to $SVG \cup SI$. Except for the program variables in S, all other variables in SVG are either known to be ground or checked for groundness and thus contain no variables. Therefore, variables can only appear in the terms to which the program variables in S and SI are bound. The success of the independence check on the pairs in SVI_g assures that none of the variables in SI is shared. Therefore, only the variables in the terms to which the variables in S are bound can be shared. By definition of the set S, these variables will not be aliased upon success of their corresponding goals (provided they weren't before) and they meet the binding conditions. However, these variables could not have been aliased before (either directly among themselves or indirectly through the variables in SI) because of the success of the checks for independence of the pairs in $(SD \cup SSI)$.■

Example 13 *: Given the collection of goals $p(f(x), g(y, z, l, m, n))$, $q(x, w, m, v)$, $r(y, h(k, n, v))$ and the global knowledge that m is ground in SS, that w and z as well as x and k are independent in SS, and that x, y meet the single, rightmost goal nv-binding and non-aliasing conditions, we have the*

sets: $SVG_g = \{x, y, n, v\}$, $S = \{x, y\}$, $SVI_g = \{(z,k),(l,w),(l,k),(w,k)\}$, $SD = \Phi$, $SI = \{z, w, k, l\}$, $SSI = \{(x,k),(y,w)\}$, $SIP = \{(w,z),(x,k)\}$. *Thus,* $SVG_{ns} = \{n, v\}$, *and* $SVI_{ns} = \{(w,k), (z,k), (l,w), (l,k), (y,w)\}$. ■

Example 14 (global correctness of CGEs w.r.t. non-strict ind.) : *Given the following goals in a difference-list quick-sort program*
`qsort(S,Sor,[P|Ls]), qsort(L,Ls,R)`
and the knowledge that s, l, p *are ground in* SS, *that* ls *is a first occurrence (and therefore independent from all other variables), and that the first qsort call nv-binds* ls, *the following CGE is globally correct w.r.t. non-strict independence:*
`(indep(Sor,R) => qsort(S,Sor,[P|LsP]) & qsort(L,Ls,R)), LsP=Ls.` ■

The type of global information needed for the construction of the sets SVG_{ns} and SVI_{ns} can be provided only by a relatively sophisticated abstract interpreter, which has to be able to detect both groundness and possible sharing in a set of variables. For example, such an abstract interpreter which can be used for non-strict as well as for strict independence is described in [17].

3.3 A Special Case of Non-Strict Independence: Negative Goals

Because of the definition of negation in Prolog as negation by failure, we can easily see that no negative literal can ever nv-bind any variable or produce any alias. In fact, even when a negative goal succeeds, this means that the corresponding positive one failed, so that any bindings created by the positive one are undone.

Let us now consider a collection of goals $g_1, \ldots, g_n$ in the body of a clause of a Prolog program, and let us suppose that some of the g_i are positive and some negative literals. Because of the above consideration about negative literals (that can be formally derived also by appropriate global analysis), the following facts hold:

- for each shared variable x, if x occurs only in negative goals or in at most one positive goal which is to the right of the negative ones, then $x \in S$;
- for each pair of non-shared variables (x, y), if both x and y occur in negative literals, or if at most one of them occurs in a positive literal to the right of the negative ones, then $(x, y) \in SIP$.

As a result of that, the following corollary holds.

Corollary 2 : *Given a collection of goals* $g_1, \ldots, g_{n-1}, g_n$, *if* g_i *is a negative goal* $\forall i = 1, \ldots, n-1$, *then they are non-strictly independent for any non-alias* θ.

Proof: If all the goals are negative, or even if g_n is positive, then each shared variable is in S and each pair of non shared variables is in SIP. So the correct *i_cond* for their non-strict independence is `true`.■

Example 15 : *Consider the following clause:*
$p(x,y,z,v,w) \leftarrow \neg r(x,v), \neg s(y,w), q(x,y), t(y,z)$.
For the four goals in the body of this clause, entered with a non-alias θ, x *is in* S. *Similarly,* (v,w), (w,z), (v,z), (x,v), (x,w), *and* (x,z) *are in* SIP. *Therefore, if no other global information is available, a globally correct* i_cond *for these four goals is simply* `ground(y)`.■

4 Conclusions

Much work has been done and is currently in progress in the compilation and implementation of IAP in its various forms. In this paper we have provided a theoretical justification to such efforts and a formal basis for the automatic generation of IAP.

We have proved the *correctness* and *efficiency* of running in parallel strictly independent goals. I.e., that the solutions obtained through parallel execution are the same as those produced by standard sequential SLD-resolution and that the total number of resolution steps is the same as in the sequential version, while the execution time is reduced. We have then introduced the concept of non-strict independence and we have shown that the same results hold for non-strictly independent goals, provided a trivial rewriting of the goals is performed, thus expanding the applicability of the method.

Most importantly, we also proposed different sets of efficient conditions which can be constructed at compile-time and then used at run-time to check for strict and non-strict independence. These different conditions apply to the cases when the goals to be executed in parallel are considered in isolation and also when they are considered as part of a clause or of a program. In this latter case we have shown how to make use of whatever clause level or program level binding information is available. Simplifications of the above conditions have also been pointed out for the interesting cases of existential variables and negative goals. In particular, we have proved that negative goals are always non-strictly independent, and that goals which share an existential variable (and one of them contains its first occurrence) are never independent. Moreover, all the proposed independence conditions have been proved to be sufficient.

The condition generation algorithms which we have presented can also be used in parallel execution methods that do not use run-time checks. In this case, it is sufficient to require that the generated compile-time condition be empty for each set of goals to be (unconditionally) executed in parallel. Furthermore, they can be used also for checking at compile- or run-time user-provided annotations.

References

[1] K. Apt. Introduction to Logic Programming. Technical Report TR-87-35, Dept. of Computer Science, The University of Texas at Austin, July 1988.

[2] K. Apt and M. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–863, July 1982.

[3] P. Biswas, S. Su, and D. Yun. A scalable abstract machine model to support limited-or restricted and parallelism in logic programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 1160–1179, 1988.

[4] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

[5] J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225, February 1985.

[6] J. S. Conery. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors. In *Symp. on Logic Prog.*, pages 457–467, August 1987.

[7] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, pages 207–229, September 1988.

[8] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[9] M. V. Hermenegildo et al. An overview of the pal project. Technical Report ACT-ST-234-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, 1989.

[10] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, U. of Texas at Austin, August 1986.

[11] M. V. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 40–55. Imperial College, Springer-Verlag, July 1986.

[12] D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming*, pages 284–297, 1988.

[13] L. Kale. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Fourth IEEE Symposium on Logic Programming*, pages 125–133. IEEE, 1987.

[14] Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[15] J. W. Lloyd. *Logic Programming.* Springer-Verlag, 1984.

[16] C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 463–475. Imperial College, Springer-Verlag, July 1986.

[17] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[18] K. Muthukumar and M. Hermenegildo. Methods for Automatic Compile-time Parallelization of Logic Programs using Independent/Restricted And-parallelism. Technical Report ACA-ST-233-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, March 1989.

[19] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.

[20] D. H. D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.

[21] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming.* MIT Press, August 1988.

[22] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.

[23] W. Winsborough and A. Waern. Transparent and- parallelism in the presence of shared free variables. In *Fifth International Conference and Symposium on Logic Programming*, pages 749–764, 1988.