STRICT AND NON-STRICT INDEPENDENT AND-PARALLELISM IN LOGIC PROGRAMS: CORRECTNESS, EFFICIENCY, AND COMPILE-TIME CONDITIONS

MANUEL V. HERMENEGILDO AND
FRANCESCA ROSSI

▷ This paper presents some fundamental properties of independent and-parallelism and extends its applicability by enlarging the class of goals eligible for parallel execution. A simple model of (independent) and-parallel execution is proposed and issues of correctness and efficiency discussed in the light of this model. Two conditions, "strict" and "non-strict" independence, are defined and then proved sufficient to ensure correctness and efficiency of parallel execution: if goals which meet these conditions are executed in parallel the solutions obtained are the same as those produced by standard sequential execution. Also, in absence of failure, the parallel proof procedure does not generate any additional work (with respect to standard SLD-resolution) while the actual execution time is reduced. Finally, in case of failure of any of the goals no slow down will occur. For strict independence the results are shown to hold independently of whether the parallel goals execute in the same environment or in separate environments. In addition, a formal basis is given for the automatic compile-time generation of independent and-parallelism: compile-time conditions to efficiently check goal independence at run-time are proposed and proved sufficient. Also, rules are given for constructing simpler conditions if information regarding the binding context of the goals to be executed in parallel is available to the compiler. ◁

Address correspondence to Manuel Hermenegildo, Facultad de Informática, Universidad Politécnica de Madrid (UPM), 28660-Boadilla del Monte, Madrid, Spain, e-mail: `herme@fi.upm.es`, or Francesca Rossi, Dipartimento di Informatica, Università di Pisa, Pisa, Italy, e-mail: `rossi@di.unipi.it`

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

1. INTRODUCTION

There has been significant interest (e.g. see [11], [22], [26], [8], [3], [5], [21], [27], [37], [38] and their references) in parallel execution models for logic programs which exploit independent and-parallelism. This type of parallelism appears to have (in common with or-parallelism – see [34] and its references for several models exploiting such parallelism) the very desirable characteristics of offering performance improvements through the use of parallelism, while at the same time preserving the conventional semantics of logic programs. However, while the correctness and potential for performance improvements of or-parallelism follows directly from the independence of the different proofs involved, results for (independent) and-parallelism are less obvious and have so far not been formally shown: there has been a need for a formal definition of goal independence and of the parallel proof procedure to be used for the execution of such goals. Also, results regarding the *correctness* and *efficiency* of such procedure have been missing. This paper attempts to fill such gaps. By correctness we refer to a combined notion of soundness and completeness of the parallel execution with respect to the standard sequential execution – i.e. to the preservation of the answer set. By efficiency, we refer to a property of the parallel execution model that determines that some performance advantage with respect to the sequential model is ensured. The importance of determining the correctness of any execution model is obvious. It is our view that efficiency results are equally important for a parallel execution model, since the fundamental objective of such a model is to reduce execution time.

Parallelism is herein understood to refer to the *simultaneous* execution of a number of sequences of resolutions by different computing agents. Exploiting parallelism then ideally means taking a computation, splitting it into "independent" threads as determined by some notion of dependency (i.e. building a dependency graph), and assigning these threads to different computing agents (both the partitioning and the agent assignment can be performed statically and/or dynamically). The need to introduce sequentialization of certain parts as determined by some criterion of dependency arises in order to preserve the correctness with respect to the sequential execution and also to ensure some notion of efficiency. In this paper we will relate the traditional concept of independence used in the aforementioned work on independent and-parallelism, which we will now call "strict" goal independence, to this objective, showing in which cases this dependency rule is sufficient to preserve the amount of work done by the sequential execution. This preservation trivially guarantees speedup if scheduling and communication overheads are ignored since simultaneous execution of the elements of a partition of a fixed amount of work is clearly guaranteed to result in a smaller total execution time than executing it sequentially.

As we will see, and due to the inherent non-determinism of logic programs, guaranteeing strict preservation of the amount of sequential work during parallel execution can be difficult in practice and in any case greatly limit parallelism. In fact, strict preservation of the amount of work is not really required in practice: intuitively it is sufficient to guarantee that in no case the parallel execution result in a longer execution time than that expected by the programmer with the sequential execution in mind. We show that this fundamental property, which we will call the "no slowdown" property, always holds for strictly independent goals, independently of whether they have solutions or not. Furthermore, we go beyond the traditional concept of strict independence and propose the more relaxed notion of "non-strict independence", showing that the correctness and efficiency (i.e. no slowdown) results also hold for this type of independence. Intuitively, this new concept considers goals independent not just if they do not share variables, but also if they do not compete for the bindings of any shared variables that might exist. This allows the parallel execution of a much larger class of goals and significantly extends the applicability of independent and-parallelism implementation technology.

Finally, while goal independence can obviously be checked at run-time, this checking can involve significant overhead. If an objective of the parallel system is the automatic generation of parallelism (and/or the verification of 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 detection of such independence at run-time. This has to be done under realistic assumptions about the binding information available to (or obtainable by) the compiler. This paper proposes efficient algorithms for compile-time generation of run-time checkable independence conditions for both strict (and, to a lesser extent) non-strict independence and proves them sufficient. It also shows how local and global binding information can be used to minimize such conditions.

These results are of direct practical application to areas such as the automatic compile-time generation of &-Prolog's Conditional Graph Expressions (CGEs) for controlling independent/restricted and-parallelism [18, 11] and reasoning about the correctness and efficiency of the bit-vector method of Lin and Kumar [27], the SDDA approach of Chang et al. [5], the Conditional Dependency Graphs and EGE generation rules of Jacobs and Langen [23], the stability rules of AKL and the underlying Extended Andorra Model [17], and other approaches based on independent-and-parallelism, as shown in Section 3.7.

The rest of the paper proceeds as follows: in Section 2 we present a simple framework for the parallel independent execution of goals and reason about its correctness and efficiency, providing the intuitions and basic results on which the notions of independence to be subsequently defined are based. In Section 3 strict goal independence is defined, and the correctness and efficiency of running in parallel strictly independent goals is shown. In sections 3.3 and 3.5 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 4 as a relaxation of the concept of strict independence. The corresponding properties are then proved and independence conditions are also given. The special and important cases of clauses which have existential variables and negative goals are treated respectively in Sections 3.6 and 4.4. Finally, in Section 6 we present our conclusions and suggest how the ideas and results of this paper can be extended to other more general frameworks (like CLP [25]), and how they can also be applied at a finer granularity level to achieve more parallelism.

2. A SIMPLE FRAMEWORK FOR INDEPENDENT PARALLEL RESOLUTION

We introduce an execution framework, which is similar to the usual sequential one in logic programming (given by SLD-resolution [28]), but where some of the goals can be *independently* run in parallel. The intuitive idea behind this parallel execution framework can be described as follows: partition the given resolvent so as to obtain some new parallel (sub)resolvents, each one associated with one of the independent goals, and a remaining part, execute the parallel resolvents in independent environments, and then embed the information gathered from such executions into the remaining part. Although such a framework is certainly not the most general one possible for describing and-parallel execution of goals, since only independent executions of the whole proof trees associated with the given goals are handled, it is nevertheless sufficient for our purposes. Because of practical reasons we will also consider, however, a slight variation of this framework where a similar partitioning into subresolvents is performed, but the execution of the subresolvents occurs in a shared binding environment. We will show that in our framework these two situations are in fact equivalent. It is certainly interesting and useful to devise more general execution frameworks which may allow parallel executions to affect each other and more flexible synchronization of goals [35]. In Section 6 we will argue that in fact it is possible to transfer our results to many such frameworks by simply applying the ideas that we will present at a different level of granularity.

Two main changes to the sequential framework are required in order to obtain the parallel framework outlined above:

- the usual sequential SLD-resolution proof procedure at each step selects only one goal in the current resolvent. Obviously, if we want to run some of the goals in this resolvent in parallel we have to allow the selection of more than one goal.
- in the sequential framework the result of the execution of one of the goals is made "visible" to the other goals by the usual notion of composition of substitutions. As we will show, such a notion is not always sufficient to express the combination of the results of the independent parallel execution of two or more goals. Thus we need to treat the case of parallel composition of substitutions specially.

Let us now describe more precisely the sequential and the parallel frameworks. The notation we will use throughout the paper follows that of Lloyd [28] and Apt [1, 2]. Moreover, in the following we consider only idempotent substitutions.

Assume that at some point of the execution $G = (g_1, \ldots, g_n)\theta$ is the current resolvent. The sequential SLD-resolution proof procedure with left-to-right selection rule would

- execute $g_1\theta$ obtaining the answer substitution θ_1 (the corresponding global substitution being $\theta\theta_1$),
- execute $g_2\theta\theta_1$, obtaining θ_2 (respectively $\theta\theta_1\theta_2$),
- execute $g_3\theta\theta_1\theta_2$, obtaining θ_3 ($\theta\theta_1\theta_2\theta_3$),
- ...

and so on until the execution of all the goals in G. Note that "executing $g_1\theta$" is normally understood as referring to the resolution of g_1 and its descendants in G until they are all resolved and g_2 appears as the leftmost goal in G. Alternatively and equivalently, and for convenience when comparing to the parallel framework, we will think of "executing $g_1\theta$" as if executing a resolvent containing only g_1 with the substitution θ applied to it. Then θ_1 represents the composition of all the most general unifiers ("m.g.u.") appearing in the resolution of $g_1\theta$ and its descendants.

In any case note that $\theta\theta_1$ represents the same substitution as would be obtained in the traditional framework, and that therefore both are equivalent.

In this framework the composition of substitutions is formally defined as follows (see [2]): consider two substitutions θ and η. Then, for any term t, $\theta\eta(t) = \eta(\theta(t))$.

If, in contrast, we want to run some of the goals in parallel, say g_i and g_j (the extension to more than two goals is straightforward), one possible execution scheme for G could be the following:

- partition G into the resolvents
  - $G_1 = (g_i)\theta$,
  - $G_2 = (g_j)\theta$,
  - $G_3 = (g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_{j-1}, g_{j+1}, \ldots, g_n)$,
- execute G_1 and G_2 in parallel obtaining the answer substitutions θ_1 and θ_2 respectively,
- apply the "parallel composition" of θ, θ_1, and θ_2 obtaining θ',
- execute $G_3\theta'$.

where we also assume that the new variables introduced during the renaming steps in the parallel execution of G_1 and G_2 belong to disjoint sets.

With this parallel framework in mind, we can restate our objective more precisely than in Section 1: we strive to run in parallel as many goals as possible within the framework presented while maintaining correctness and efficiency with respect to the sequential execution. In other words, we assume that the answers obtained by the sequential execution (with a left-to-right selection rule) correspond to the intended model of the program, and that we would like to preserve such answers in the parallel execution while improving on the time taken to obtain them by the sequential execution, i.e. we would like to obtain the answers possibly in a shorter time, but certainly not in longer time, thus ensuring that the "no slowdown" property introduced in Section 1 holds (throughout the paper we use the concept of "time" to refer to the number of steps in an execution).

2.1. *Correct Parallel Composition*

One issue that must be taken into account is how the abstract notion of "parallel composition" is defined. In fact, if we use the above described standard (sequential) composition of substitutions we may obtain incorrect results, as shown by the following example. Consider the resolvent
:- $p(x), q(x)$.
and the following definition of p and q:
$p(a)$.
$q(b)$.

In this case, the sequential execution framework first executes $p(x)$, returning $\{x/a\}$ and the new resolvent :- $q(x)\{x/a\}$, i.e. :- $q(a)$, whose execution fails, thus making the whole given resolvent fail. On the other hand, the parallel execution framework executes in parallel $p(x)$ and $q(x)$, returning $\{x/a\}$ and $\{x/b\}$ respectively. Note now that the composition of $\{x/a\}$ and $\{x/b\}$ is, according to the definition in [28, 1, 2], the substitution $\{x/a\}$. Thus we obtain a different answer. In this simple example it is easy to realize that the problem is due to the sharing of the variable x which both the p and q goals try to instantiate. However, incorrect

answers can be obtained even when there is no conflicting binding for the shared variables. Let us consider the following example, where we have the resolvent
:- $p(x, y), q(y)$
and the following definitions of p and q:
$p(z, z)$.
$q(a)$.
If we run $p(x, y)$ and $q(y)$ sequentially, we first execute $p(x, y)$ returning $\theta_p = \{x/z, y/z\}$, and then we execute $q(y)\theta_p$, i.e $q(z)$, returning $\theta_q = \{z/a\}$. Thus, in the end, we obtain the substitution $\theta = \theta_p\theta_q = \{x/a, y/a, z/a\}$. If we now execute $p(x, y)$ and $q(y)$ in parallel, we obtain θ_p from the execution of $p(x, y)$, and $\theta'_q = \{y/a\}$ from the execution of $q(y)$, thus ending with their composition $\theta_p\theta'_q = \{x/z, y/z\}$ or $\theta'_q\theta_p = \{y/a, x/z\}$ as final substitution, that is obviously different from the θ obtained from the sequential execution, and thus again an incorrect result.

One possible way to avoid the possibility of such incorrect results is to adapt the definition of composition of substitutions to the cases when we have to compose the results of some parallel executions. More precisely, the definition of "parallel composition" could be as follows: consider again two substitutions θ and η, and their representations as sets of equations, $E(\theta)$ and $E(\eta)$. Then, given any term t, their parallel composition is $\theta \circ \eta(t) = mgu(E(\theta) \cup E(\eta))(t)$.

In the first of the examples above, this new definition would fail while computing the composition of $\{x/a\}$ and $\{x/b\}$, because there is no m.g.u. for $x = a$ and $x = b$. Thus the results of the sequential and the parallel execution would be the same.

In the second example, the composition of $\{x/z, y/z\}$ and $\{y/a\}$ would be computed, according to the new definition, to be $mgu(\{x/z, y/z\} \cup \{y/a\}) = \{x/a, y/a, z/a\}$. Thus, again the answer substitutions of the sequential and the parallel execution coincide.

2.2. A Parallel Framework using Standard Composition

It is interesting to note that if $var(\theta) \cap var(\eta) = \emptyset$, then $mgu(E(\theta) \cup E(\eta))(t) = \eta(\theta(t))$, i.e. $(\theta \circ \eta)(t)$. In other words, parallel composition coincides with sequential composition when the goals to be run in parallel do not share any variable. In fact, if two goals share no variables, then also their answer substitutions share no variables. This observation is relevant, since the adoption of a new definition of composition would require a revision of well known results in logic programming, which rely on the standard definition. Since this is beyond the scope of this paper, we will instead adopt a different but equivalent approach herein: we will transform any set of goals to be executed in parallel into one where no variables are shared, in such a way that soundness (with respect to the given resolvent) is preserved, generality is not sacrificed, and the usual definition of composition, with all the well-known results which follow from it, can be used.

The transformation that we consider involves eliminating any shared variables in goals which are to be executed in parallel by renaming all their occurrences (so that no two occurrences in different goals have the same name), and adding some unification goals to reestablish the lost links. More precisely:

Definition 1. (renaming transformation) A renaming transformation, applied to a sequence of goals $G = g_1, \ldots, g_n$, and a substitution θ, is a rewriting of G

into the sequence $G' = g'_1, \ldots, g'_n, R$, defined as follows. Let $occ(x, g)$ be the set of all occurrences of variable x in goal g. Let x denote any variable shared by two or more $g_i\theta$, and let $g_x\theta$ denote the leftmost goal containing that x. Each g'_i is a renaming of $g_i\theta$ such that for every x, all the occurrences of x are renamed except those contained in $g_x\theta$ and the renaming is performed in such a way that, for every g'_i, all the occurrences of x in $occ(x, g_i\theta)$ have the same name and, given g'_i and g'_j, the occurrences of x in $occ(x, g_i\theta)$ have a name different from the name of those in $occ(x, g_j\theta)$. R is the sequence of goals formed by all goals of the form $x = x'$ for every new variable x' introduced in the renaming process of x.

Example 1. Consider the collection of goals ($r(x, z, x)$, $s(x, w, z)$, $p(x, y)$, $q(y)$) in a resolvent (we consider θ already applied to the goals). According to the definition of renaming transformation, we will write this new collection of goals as follows:
$r(x, z, x), s(x', w, z'), p(x'', y), q(y'), x = x', x = x'', y = y', z = z'$.

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

Goals of the form $x = x'$ above are called "back-binding" goals, and are related to the back-unification goals defined in [26], and the closed environment concept of [8]. Note that the new resolvent is logically equivalent to the given one since the unification goals simply make some bindings explicit.

Let us consider again the two examples above. In the first one, after the transformation we would have
:- $p(x), q(x'), x = x'$.
$p(a)$.
$q(b)$.

Thus the parallel execution of $p(x)$ and $q(x')$ produces $\{x/a\}$ and $\{x'/b\}$, whose composition is (usual definition) $\{x/a, x'/b\}$. After that, we are left with the resolvent :- $(x = x')\{x/a, x'/b\}$, i.e. :- $a = b$, which fails, as in the sequential execution.

For the other example, we have:
:- $p(x, y), q(y'), y = y'$.
$p(z, z)$.
$q(a)$.

Here the parallel execution of $p(x, y)$ and $q(y')$ produces $\{x/z, y/z\}$ and $\{y'/a\}$, whose composition is (usual definition) $\{x/z, y/z, y'/a\}$. After that, we execute the resolvent :- $(y = y')\{x/z, y/z, y'/a\}$, i.e. :- $(z = a)$, which returns $\{z/a\}$. Thus the final answer substitution is the composition of $\{x/z, y/z, y'/a\}$ and $\{z/a\}$, i.e. $\{x/a, y/a, y'/a, z/a\}$, which coincides with the answer substitution obtained by the sequential execution (if projected on the same variables).

Thus, for the rest of this paper, we will use the above described renaming transformation whenever we want to execute in parallel goals which share variables. Therefore, the parallel execution framework proposed at the beginning of this section is now transformed as follows:

Given a resolvent $G = (g_1, \ldots, g_n)\theta$ and the knowledge that $g_i\theta$ and $g_j\theta$ are to be executed in parallel the following steps are to be performed:

- Apply the renaming transformation to $g_i\theta, g_j\theta$.
- Assuming that the result of the renaming transformation above is g'_i, g'_j, R construct the following resolvents:
  - $G_1 = (g'_i)$,
  - $G_2 = (g'_j)$,
  - $G_3 = R$,
  - $G_4 = (g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_{j-1}, g_{j+1}, \ldots, g_n)$.
- Execute G_1 and G_2 in parallel.
- Assuming θ_1 and θ_2 are the answer substitutions obtained in the previous step, execute $G_3\theta_1\theta_2$.
- Assuming θ_3 is the answer substitution obtained in the previous step, execute $G_4\theta\theta_1\theta_2\theta_3$.

In the above it is assumed that the new variables introduced during the renaming steps in the parallel execution of G_1 and G_2 belong to disjoint sets. Also, note that the parallel framework can be applied recursively within the execution of G_1 and G_2 in order to allow nested parallelism.

It is important to say at this point that, while the framework just proposed can handle the parallel execution of non-consecutive goals, in this paper we will always consider collections of consecutive goals. The reason for this will become obvious in the following sections. However, it may be informally justified by saying that the choice of the collection of goals to run in parallel is usually made so as to meet certain requirements (like correctness and efficiency, as we will see later), and that it is much easier to check such requirements if the goals are consecutive.

As a final observation, note that in the parallel execution framework G_1 and G_2 are assumed to execute in different environments and θ_1 and θ_2 remain separate. From a practical point of view this quite accurately reflects the actual situation in distributed implementations of independent and-parallelism [8]. However, in models designed for shared addressing space machines the goals executing in parallel generally share a single binding environment (e.g. [21, 18, 27]). Note, however, that after the renaming transformation no variables are ever shared among the resolvents being executed in parallel. Thus, bindings performed during the execution of one of these resolvents cannot be "seen" by the other and vice-versa. Therefore, in practice the latter situation is essentially identical to that of separate environments and both types of implementation can be seen as equivalent when implementing the renaming transformation framework described in this section. In fact, although introducing the shared variable renaming transformation has been justified from the point of view of allowing the use of the standard composition of substitutions, the isolation of environments resulting from such renaming transformation is of practical importance and an additional powerful reason to perform the transformation, as will be shown later.

2.3. Failure Handling

An important issue which remains to be discussed is how failure is handled in the parallel framework. Note that in the sequential framework (with depth-first search rule) a failure means simply returning to the last point where a choice was made and taking an alternative branch of the proof tree. In the parallel framework this still holds within the execution of each of the resolvents being considered. A special case occurs however when no answer can be found for one of the resolvents being executed in parallel (G_1 or G_2). The framework assumes that at such a point all the computation associated with the other resolvent (respectively G_2 or G_1) is interrupted and control returns to the last choice point before the parallel execution of G_1 and G_2. If it is the execution of G_3 or G_4 that has no solutions then the standard backtracking algorithm is used, with the choice points of G_2 being considered in reverse order and before those of G_1, as in the sequential execution. However, when returning to a choice point in G_1, two alternatives are possible [22]. In the first one, referred to as "point-backtracking", first the next solution for G_1 is computed and then, after it is found, G_2 is restarted. In the second one, referred to as "streak-backtracking", execution of G_2 is (re)started in parallel with the computation of the next solution for G_1. This allows more parallelism but also has a greater potential for performing unnecessary work. In the following we will assume "point-backtracking".

It should be noted that in the framework proposed if there is more than one successful branch in the search tree of G_1, G_2 is *recomputed* for each such branch, as in the sequential model [22]. A possible alternative to this failure rule implies gathering all possible answers for G_1 and all possible answers for G_2 when running them in parallel, thus computing each such answer only once [26, 16]. Failure behavior (i.e. if one of the two goals has no solutions) would imply interrupting the parallel computation of the other goal and returning to the first choice point before the parallel conjunction. Both of these approaches have their merits and drawbacks. The former allows easier implementation and requires less memory, since only one binding environment needs to be kept at each point in time. The latter sometimes allows saving computation. In practice a non-recomputing behavior can be implemented in a system which uses recomputation through the use of "all solutions" predicates. Throughout the paper we will generally assume the framework as proposed in this section, i.e. using recomputation, because it represents the worst case regarding generation of additional work and thus the results obtained bound those applicable to the "non-recomputing" version of the framework. However, we will also present more specific results for both models in some cases where the differences are of special interest.

2.4. Correctness Issues

As mentioned before, by correctness we refer herein to a combined notion of soundness and completeness of the parallel execution with respect to the standard sequential execution – i.e. to the preservation of the answer set. We recall that if the goals to be run in parallel share variables then they will be renamed to eliminate the sharing before their parallel execution. Thus the correctness comparison should be between that of the sequential execution of the goals and that of the parallel execution of their renamed versions, plus the execution of the back-binding goals.

If the goals that are executed in parallel are "pure" and they do not fail then the parallel execution framework described at the end of the previous section is obviously sound with respect to the sequential one, due to the equivalence of the resolvent after the renaming transformation, and to the use of the standard composition of substitutions. If failure occurs it is also obvious that soundness is preserved: within each resolvent, execution proceeds in the same way as in the sequential model (if nested parallelism occurs, then the algorithm is applied recursively and its correctness also follows by induction). In the special case in which no answer can be found for G_1, execution is also identical to that of the sequential framework, except for discarding G_2. This is clearly sound since G_2 would not be executed at all in the sequential framework. If no answer can be found for G_2 then, since due to the renaming transformation it has no variables in common with G_1, it is clear that no solution would be found for G_2 in the sequential execution either and independently of G_1. In fact, given that the goals in G_2 in the sequential execution could only have been more instantiated and since G_2 contains only pure goals the sequential execution would also have failed. Thus, it is correct to discard G_1 and backtrack to the previous choice point before G_1 and G_2. Thus, the execution model proposed is sound for pure goals.

On the other hand, completeness (again with respect to the sequential framework) is not preserved. In fact, the transformation may in general turn a finite behavior into an infinite behavior. Consider for example: :- $p(x), q(x)$.
$p(a)$.
$q(b)$:- $q(b)$.
$q(a)$.
In this case the sequential execution terminates in finite time with answer substitution $\{x/a\}$. The proposed renaming transformation of the resolvent would result in
:- $p(x), q(x'), x = x'$.
The parallel execution loops infinitely. Thus, it is clear that only certain classes of goals can be parallelized while preserving completeness and that some sufficient conditions will have to be found in order to identify such goals. To this end it is important to note that the leftmost goal (i.e., $p(x)$) binds the shared variable x in the sequential execution and therefore prunes the search space of the rightmost goal, and that the rewriting prevents this pruning from happening. Furthermore, the rightmost goal also binds the shared variable, possibly to a different value.

Furthermore, programs in practice often contain extra-logical predicates and this causes additional problems, so that not only completeness but even soundness may be affected. Two extra-logical predicates of interest are $var/1$ and ! (cut). Consider the following example:
:- $p(x), q(x, y)$.
$p(a)$.
$q(x, y)$:- $var(x), !, y = a$.
$q(x, y)$:- $y = b$.
Again, the proposed renaming transformation of the resolvent would result in
:- $p(x), q(x', y), x = x'$.
and the parallel execution would succeed with $\{x/a, y/a\}$ while the sequential execution would succeed with $\{x/a, y/b\}$. Thus, it is clear that only certain classes of impure goals can be parallelized while preserving soundness and further conditions will have to be found in order to identify such goals.

Finally, another class of goals whose parallel execution can create differences in observed behavior with respect to the sequential execution are those containing side effects such as, for example, input/output predicates. For simplicity, and since this subject has been treated at length elsewhere [13, 30, 6] and the solutions are orthogonal and compatible with those presented herein, we will consider it outside the scope of this paper. Alternatively, the preservation of side effect behavior in parallel execution could also be considered as an additional constraint on the class goals which can be executed in parallel which would yield different parallelization conditions from those that will be proposed herein.

2.5. *Efficiency Issues*

Preserving correctness is not sufficient in general, since we will be interested also in the issue of *efficiency* of the parallel execution. As mentioned in the introduction, efficiency can be understood at two levels: one is preservation of the amount of work, which ensures speedup (modulo scheduling and communication overheads). The other, more lax, requirement is to simply ensure that the "no slowdown" property holds, i.e. that parallel execution time is guaranteed to be shorter or equal than sequential execution time. Clearly if the parallel execution requires more time than the sequential one, then the very aim of parallel computation would be defeated.

Again we recall that, if the goals to be run in parallel share variables, then they will be renamed to eliminate such sharing before their parallel execution. Thus the efficiency comparison should be between that of the sequential execution of the goals and that of the parallel execution of their renamed versions, plus the execution of the back-binding goals. In fact, the execution of the back-binding goals themselves generally represents a small amount of work and arguably can be ignored at the granularity level of our comparisons: it can be considered one step at most since any sequence of back-bindings $x_1 = x_1', \cdots, x_n = x_n'$ can be encoded in a single unification $t(x_1, \cdots, x_n) = t(x_1', \cdots, x_n')$ where t is any functor. However, other effects related to the renaming transformation and the back-binding goals have to be taken into account: sometimes the parallel execution of any set of (consecutive) goals which share variables (before the renaming transformation) could, although being correct, result in an increase of the work involved and/or of the execution time quite unrelated to the execution of the back-binding goals themselves.

Consider the following example:

:- $p(x), q(x)$.
$p(a)$.
$q(b)$:- $proc$.

where $proc$ is very costly to execute.
The renamed resolvent is:
:- $p(x), q(x'), x = x'$.

Both the sequential and the parallel execution fail. However, their efficiency is quite different. The sequential execution of $p(x), q(x)$ executes $p(x)$ returning $\{x/a\}$, and then fails in trying to match $q(x)\{x/a\}$, i.e. $q(a)$, to any rule head. It thus never goes into the execution of $proc$. In contrast, the parallel execution executes in parallel $p(x)$ and $q(x')$ returning $\{x/a\}$ and $\{x'/b\}$ (but only after executing $proc$ too), and then fails in the execution of the goal $x = x'\{x/a\}\{x'/b\}$, i.e. $a = b$. Thus the sequential execution is much more efficient and, if the execution time of $proc$ is large, the parallel execution could take much longer than the sequential one.

The cause of the difference in efficiency in this example, and also in general, is that the execution of $p(x)$ affects the execution of $q(x)$ in the sequential framework, thus restricting the search space of $q(x)$. As mentioned before, in our parallel framework parallel executions will not affect each other and thus the search space of $q(x)$ is bigger. It could be argued that if a parallel goal is allowed to affect another by running both in the same environment (i.e. sharing the bindings being

performed), this problem is eliminated. However, apart from the otherwise solvable complications regarding the definition of parallel composition of substitutions, this results in other problems. Consider the following example:
:- $p(x), q(x)$.
$p(a) : -fail.$
$p(X)$.
$q(b)$.
where no rewriting of the resolvent is performed. The sequential execution will succeed in three steps, $q(b)$ being executed only once. On the other hand the binding of x to a done by p before failing may make $q(b)$ fail creating either a wrong result or, at least, a complex backtracking in which $q(b)$ has to be restarted after p backtracks over the binding of x to a. On the other hand, if the resolvent is renamed as in the proposed model, thus implementing the separation of environments, parallel execution will perform the same number of steps as the sequential execution.

In previous examples the restriction of the search space of $q(x)$ is due to the instantiation of a shared variable (i.e., x) by $p(x)$. However, this is not the only way a search space can be restricted. In fact, aliasing of previously unaliased shared variables can achieve the same effect. Consider the following example:
:- $p(x, y), q(x), r(y)$.
$p(z, z)$.
$q(a)$.
$r(b)$:- $proc$.
where again $proc$ is very costly to execute. The sequential execution first executes $p(x, y)$ returning $\{x/z, y/z\}$, then $q(z)$ returning $\{z/a\}$ and then fails in trying to match $r(a)$. In contrast, the parallel execution executes in parallel $p(x, y)$, $q(x')$ and $r(y')$. Thus $r(y')$ can match with $r(b)$ and this leads to the execution of $proc$.

It is worth noticing that in the two examples above where the renaming transformation was performed the back-binding goals fail. Intuitively, this means that the sequential executions of the goals affected each other in some way, but, since the link between them, which is captured by the back-binding goals, is lost during their parallel executions, failure can only be detected at back-binding time. Thus, in order to ensure efficiency, we need to avoid such cases and this at least means ensuring that back-binding goals never fail.

However, simply ensuring that the back-binding goals do not fail is not sufficient. Consider the following program:
:- $p(x), q(x)$.
$p(a)$.
$q(b)$:- $proc, fail$.
$q(a)$.
where again $proc$ is very costly to execute. In this case the sequential execution first executes $p(x)$ returning $\{x/a\}$, and then executes $q(a)$ successfully. The parallel execution, instead, executes $p(x)$ and $q(x')$ returning $\{x = a\}$ for both, but executing all of $proc$ before failing, and thus going to the second rule whose head unifies with $q(x')$. The cause in this case is again that the leftmost goal provides a binding that prunes the search space of the rightmost goal and the renaming transformation eliminates such pruning.

One idea in order to avoid the inefficiencies created in all the cases above (and also the correctness problems pointed out in the previous section) is to only parallelize if the goal to the left simply does not affect in any way the goal to the right in the sequential execution. This certainly holds if the rightmost goal in the parallel case is identical (i.e. executes in the presence of the same substitution on its variables) to that of the sequential case, perhaps modulo renaming of some variables. This corresponds to the notion of "call instantiation correctness" given by Winsborough and Waern [38]. In that case, the "shape" of the execution tree corresponding to the goal to the right will be preserved with respect to the sequential execution (although perhaps its nodes are slightly different due to, for example, the renaming of variables). The following results simply express this more formally. Let us denote by $t_1 \equiv t_2$ the fact that t_1 and t_2 are identical modulo variable renaming.

Theorem 1. (preservation of sequential execution) Consider two goals g_1 and g_2 and their renamed versions g_1' and g_2' for θ, according to the renaming transformation. Let θ_1 be an answer substitution of $g_1\theta$. Assume also that for any θ_1, $g_2\theta\theta_1 \equiv g_2\theta$. Then we have that the search trees of $g_1\theta$ and $g_2\theta\theta_1$ have the same shape as those of g_1' and g_2' respectively.

Proof. By definition of the renaming transformation we have that all new variables introduced by the renaming are in g_2'. Thus $g_1\theta = g_1'$. Therefore it is clear that the search tree of $g_1\theta$ has the same shape as that of g_1'. On the other hand, if $g_2\theta \equiv g_2\theta\theta_1$, by the properties of SLD resolution, it is obvious that the search tree of $g_2\theta\theta_1$ will have also the same shape as that of $g_2\theta$. Since $g_2\theta \equiv g_2'$ also holds, and following the same reasoning, the shape of the corresponding trees is also identical.□

Note that, as a corollary of the above theorem we can say that if the above conditions are satisfied the time (number of steps) involved in the sequential execution of $(g_1, g_2)\theta$ is the same as that involved in the *sequential* execution of g_1', g_2'.

The following result also holds if the conditions of Theorem 1 hold:

Theorem 2. (success of back-bindings) Consider two goals g_1 and g_2, their renamed versions g_1' and g_2' for θ, , according to the renaming transformation, and the set R of back-bindings resulting from the renaming transformation. Let θ_1 be any answer substitution from the execution of $g_1\theta$. Then, if for any θ_1, $g_2\theta\theta_1 \equiv g_2\theta$, then, when g_1', g_2', R is executed, all the goals in R will succeed.

Proof. The goals in R are of the form $x_i = x_i'$ where the x_i are distinct shared variables between $g_1\theta$ and $g_2\theta$. Let us reason about the instantiation state of the x_i after the execution of g_1' and g_2'. First, it is clear that the execution of g_2' cannot affect the x_i since those variables do not occur in g_2'. Regarding g_1' we have that $g_1' = g_1\theta$. By hypothesis, for any θ_1, $g_2\theta\theta_1 \equiv g_2\theta$ and, thus, $g_2\theta\theta_1 \equiv g_2'$ (since g_2' is itself a variant of $g_2\theta$). For this to be true, θ_1 could rename the x_i but it would have to leave them as distinct variables. Thus, we have that after the execution of g_1' and g_2' the x_i will be free and distinct variables and therefore, independently of the instantiations of the x_i', all the back-binding goals will be simple variable bindings and trivially succeed. □

Another idea in order to avoid the possible inefficiencies of parallel execution illustrated in this section is based on the observation that, if a goal is pure, and a step in its computation does not modify the state of a given variable, then an equivalent step would also be performed in another binding context which is otherwise identical but in which that variable is further instantiated (note that this may not be the case for impure goals, in particular if they include calls to predicates in the class of "var/1"). As an example, consider the following resolvent:

:- $p(x), q(x, y)$.
with current substitution $\theta = \{y/b\}$ and the following program:
$p(a)$.
$q(x, b)$.
$q(x, c)$.
It is clear that it is safe to execute both goals in the resolvent in parallel since the binding produced by p cannot affect the execution of the pure goal q (which does not "touch" x). Note that this also holds for the following definition of q:
$q(x, b)$.
$q(d, c)$.
More formally:

Theorem 3. (preservation of sequential execution in case of pure goals) Consider two goals g_1 and g_2 and their renamed versions g_1' and g_2' for θ, according to the renaming transformation. Assume that g_2 is a pure goal. Let W be the set of variables defined as $W = \{x \in vars(g_2\theta) | \exists \theta_1$ answer of $g_1\theta, \exists y \in vars(g_2\theta), s.t.$ $\{x, y\}\theta_1 \not\equiv \{x, y\})\}$. Assume also that for any partial answer μ of $g_2\theta$, W is identical (modulo variable renaming) to $W\mu$ ($W \equiv W\mu$). Then we have that the search trees of $g_1\theta$ and $g_2\theta\theta_1$ have the same shape as those of g_1' and g_2' respectively.

Proof. By definition of the renaming transformation we have that all new variables introduced by the renaming are in g_2'. Thus $g_1\theta = g_1'$. Therefore it is clear that the search tree of $g_1\theta$ has the same shape as that of g_1'. On the other hand, if g_2 is a pure goal and there exists a set of variables $V \subseteq vars(g_2\theta)$ such that, for any μ, $V \equiv V\mu$, for any θ' such that $domain(\theta') \cap vars(g_2\theta) \subseteq V$ the shape of the trees of $g_2\theta$ and $g_2\theta\theta'$ will be the same. In particular this is satisfied by any answer θ_1 of $g_1\theta$ since by definition $domain(\theta_1) \cap vars(g_2\theta) \subseteq W \subseteq V$. Furthermore, the renaming η such that $g_2\theta\eta = g_2'$ also satisfies the conditions. Thus, the shape of the trees of $g_2\theta$, $g_2\theta\theta_1$, and g_2' will be the same.□

The following result also holds if the conditions of Theorem 3 hold:

Theorem 4. (success of back-bindings in case of pure goals) Consider two goals g_1 and g_2, their renamed versions g_1' and g_2' for θ, , according to the renaming transformation, and the set R of back-bindings resulting from the renaming transformation. Assume that g_2 is a pure goal. Let W be the set of variables defined as $W = \{x \in vars(g_2\theta) | \exists \theta_1$ answer of $g_1\theta, \exists y \in vars(g_2\theta), s.t. \{x, y\}\theta_1 \not\equiv \{x, y\})\}$. Assume also that for any partial answer μ of $g_2\theta$, $W \equiv W\mu$. Then, if g_1', g_2', R is executed, all the goals in R will succeed.

Proof. The goals in R are of the form $x_i = x_i'$ where the x_i are distinct shared variables between $g_1\theta$ and $g_2\theta$ and x_i' are variables in g_2'. First, it is clear that the execution of g_2' cannot affect the x_i since those variables do not occur in g_2'. Also regarding g_1' by definition of renaming transformation we have that $g_1' = g_1\theta$. By hypothesis, for set W of x_i' affected by g_2', for any answer θ_1 of g_1', $W\theta_1 \equiv W$, and therefore $x_i = x_i'$ will succeed. For the rest of the x_i' variables which remain unbound and distinct after the execution of g_2', it is also clear that $x_i = x_i'$ will also succeed.□

It is easy to show by induction that the results above also hold for any number of goals.

2.6. Parallel Efficiency

The results of Theorems 1 and 3 allow us to compare the *sequential* execution of $(g_1, g_2)\theta$ and the *sequential* execution of g'_1, g'_2 and see that they have the same shape and thus the same number of steps for corresponding paths of the search if the conditions of the theorems hold. However, clearly what we are really interested in comparing is the sequential execution of $(g_1, g_2)\theta$ with the parallel execution of g'_1, g'_2. In doing that the following result is instrumental:

Theorem 5. (equivalence of succeeding branches) Consider two goals g_1 and g_2, and their renamed versions g'_1 and g'_2 for θ, according to the renaming transformation. Assume these goals meet the conditions of Theorem 1 or Theorem 3. Then, any non-failing branch of the sequential execution tree of $(g_1, g_2)\theta$ and the concatenation of the corresponding branches in the parallel execution of g'_1, g'_2 have the same number of steps.

PROOF. By Theorems 1 and 3 any branch of the sequential execution tree of $(g_1, g_2)\theta$ and the concatenation of the corresponding ones of the sequential execution of g'_1, g'_2 have the same number of steps. Since g'_1, g'_2 have no variables in common, their executions cannot affect each other (note that we have ruled out side effects), except for their interaction through the failure rule, which is ruled out by hypothesis. Thus, two corresponding, non-failing branches of the sequential and parallel execution trees of g'_1, g'_2 when concatenated also have the same number of steps. □

Theorems 1 and 3 also guarantee the equivalence of the sequential execution of $(g_1, g_2)\theta$ and (g'_1, g'_2) for failing branches. However, while the failure rule of section 2.3 does guarantee that if failure occurs in the sequential execution, it will also occur in the parallel execution, it does not prevent it from happening at possibly different stages of unfolding of the parallel and sequential trees. As we will show, in these cases, both more and less steps can be performed than in the sequential execution, but the "no slowdown" property will still be preserved.

In the following we will assume for simplicity an unbounded number of processors. If the number of processors is limited then, provided a "left biased" scheduling strategy is used (i.e. one which guarantees that at least the leftmost goal in syntactic order is selected for execution), the results will still be bounded from above by those for the sequential execution and from below by those for the parallel execution with an unbounded number of processors. We will also assume an ideal situation where no overhead due to parallelism is incurred. Thus, the results obtained will only be applicable to a practical implementation to the extent that such parallelism overheads are sufficiently low in that implementation. However, achieving quite low overhead appears to be attainable in practice in most cases as demonstrated by systems such as &-Prolog/RAP-WAM [21, 18], and APEX [27].

Let us assume that the tree associated with $g_1\theta$ and g'_1 (respectively $g_2\theta\theta_1$ and g'_2) has m_1 (respectively m_2) solutions, and that k_1^i (respectively k_2^i) steps are executed between the $(i-1)$th. and the ith solution (including intermediate backtrackings, i.e. the steps leading to intervening failures). The last of these costs $(k_1^{m_1+1}, k_2^{m_2+1})$ represent the time needed to detect that there are no more solutions for the goal. We also define $k_1 = \sum_{i=1}^{m_1+1} k_1^i$ (respectively $k_2 = \sum_{i=1}^{m_2+1} k_2^i$). Let us also call W_s (respectively W_p) the work involved in the sequential (respectively parallel) execution, and T_s (respectively T_p) the time of such execution. As mentioned before, there are at least two interesting conceptual models for how the gathering of alternatives is handled in parallel depending on whether recomputation is performed or not. We will derive expressions for W_p and T_p for both models. W_p^n and T_p^n will represent work and time in the model with no recomputation, i.e. where we assume that all the m_1 and m_2 branches are computed independently and then joined [26, 16]. W_p^r and T_p^r will represent work and time for the alternative approach of recomputing each of the m_2 solutions of g_2' for each of the m_1 solutions of g_1' (as in the sequential model and in [21]). We will consider a number of cases.

2.6.1. Case 1: Both goals have one or more answers. Let us first assume that both of the two goals have at least one answer. Then we have that

$$\begin{array}{lcl}
W_s & = & \sum_{i=1}^{m_1}(k_1^i + k_2) + k_1^{m_1+1} = k_1 + m_1 k_2 \\
T_s & = & W_s \\
W_p^n & = & k_1 + k_2 \\
T_p^n & = & \max(k_1, k_2) \\
W_p^r & = & (k_1^1 + k_2^1) + \sum_{j=2}^{m_2+1} k_2^j + \sum_{i=2}^{m_1}(k_1^i + k_2) + k_1^{m_1+1} \\
 & = & \sum_{i=1}^{m_1}(k_1^i + k_2) + k_1^{m_1+1} = W_s \\
T_p^r & = & \max(k_1^1, k_2^1) + \sum_{j=2}^{m_2+1} k_2^j + \sum_{i=1}^{m_1}(k_1^i, k_2) + k_1^{m_1+1} \\
 & \leq & \sum_{i=1}^{m_1}(k_1^i + k_2) + k_1^{m_1+1} = T_s
\end{array}$$

It is straightforward to see that $T_p^n \leq T_s$ in any case, since $\max(k_1, k_2) \leq k_1 + k_2 \leq k_1 + m_1 k_2$, since we have assumed $m_1 \neq 0$. As for the comparison of W_s and W_p^n, it depends on the value of m_1. However, in the worst case (i.e., when $m_1 = 1$), we have that $W_p^n = W_s$, and thus in general $W_p^n \leq W_s$. We have also shown that $W_p^r = W_s$ and that $T_p^r \leq T_s$. Thus, if both goals have at least one solution, the time and work are always less or equal, both for the non-recomputation (as shown by [26]) and for the recomputation cases.

2.6.2. Case 2: At least one goal finitely fails. Let us now consider the case in which at least one of the goals finitely fails, i.e. it has no answers, and this can be determined in finite time. We need to distinguish between case when g_2 fails and that when g_1 fails, since this makes a clear difference in the sequential execution. In the first case the sequential execution with the left-to-right selection rule entirely (and unnecessarily) executes g_1 and then g_2 until its failure. In contrast, the parallel execution runs g_1 and g_2 in parallel until the failure of g_2, given the communication of failure among processors stipulated in the parallel framework [22], possibly executing a smaller part of g_1. Thus, we have

$$\begin{array}{lclcl}
W_s & = & T_s & = & k_1 + m_1 k_2 \\
W_p^r & = & W_p^n & = & k_2 + \min(k_1, k_2) \\
T_p^r & = & T_p^n & = & k_2
\end{array}$$

In the second case –when g_1 fails– the sequential execution following the left-to-right selection rule starts backtracking immediately after the failure of g_1, without executing g_2 at all. In contrast, the parallel execution runs both goals until the failure of g_1, thus (unnecessarily) executing part or all of g_2. More precisely, we have

$$\begin{array}{lclcl}
W_s & = & T_s & = & k_1 \\
W_p^r & = & W_p^n & = & k_1 + \min(k_1, k_2) \\
T_p^r & = & T_p^n & = & k_1
\end{array}$$

Thus, in any of the two cases above we have $T_p \leq T_s$ (of course the most fortunate situation for the parallel execution is when g_2 fails). Thus, the time is always less or equal, while the work may be more (up to twice as much). Therefore, the "no slowdown" property is preserved. However, it appears important to detect failing goals ahead of time in order to avoid wasted (speculative) work. This result shows that in general it will always be more desirable to run two goals in parallel when it is known that the leftmost one has at least one solution.

2.6.3. Case 3: Infinitely failing goals. In case of infinite failure of the sequential execution, it is possible for the parallel execution to finitely fail. Thus the finite failure set of the parallel execution model could be larger than that of the sequential model. This could be rephrased by saying that the selection rule [28] used in parallel execution is "fairer" than the sequential one. As an example, let us consider the goal
:- $p(x), q(y)$.
and the program
$p(x)$:- $p(x)$.
In this case, the sequential execution would infinitely fail in the execution of $p(x)$, while the parallel execution would finitely fail due to the finite failure of $q(y)$. Clearly, $W_p < W_s$ and $T_p < T_s$ hold for both modes.

2.7. Efficient Parallel Execution

Theorem 6. (efficient parallel execution) Consider two goals g_1 and g_2, their renamed versions g_1' and g_2' for a substitution θ, and the set R of back-binding goals generated during the renaming transformation. Let us also call T_s the time for the sequential execution of $(g_1, g_2)\theta$ and T_p the parallel execution time for g_1' and g_2'. Assume also that either the conditions of Theorem 1 or those of Theorem 3 hold. Then we have that $T_p \leq T_s$.

Proof. Follows directly from Theorems 1 and 2 or 3 and 4, Theorem 5, and the discussion in the previous sections comparing the parallel and sequential execution time.□

This result defines two classes of goals for which the "no slowdown" property can be ensured. Thus, we may conclude that a general pragmatic solution to solve the efficiency problems described in this section is to run in our parallel framework only those sets of goals that are in such classes. An immediate consequence of the conditions on the goals in such classes is that we can safely run in parallel goals that have no variables in common. Alternatively, two goals can also be run in parallel in the case in which only the second goal is allowed to instantiate any shared variables, or in the case in which, if the first goal instantiates any shared variables, then the second one is pure and it does not "touch" such variables. The first situation – which corresponds to the traditional concept of independence referred to in the introduction [8, 11, 21]– will be called "strict independence", while the second one corresponds to the concept of "(generalized) non-strict independence" introduced in [20], as a formalization and generalization of the early ideas of [11, 36, 38]. In the following sections, we will discuss more formally the classes of strictly and non-strictly independent goals and the properties of their parallel execution, and we will give some sufficient conditions, to be generated at compile time, whose success at run-time will guarantee that a collection of goals belongs to one of such classes.

3. STRICT GOAL INDEPENDENCE

In this section we present the usual concept of independence of a set of goals, as introduced by [8, 11, 21], which, as mentioned before, and following [19], we will from now on refer to as "strict independence." We prove that if such a set is strictly independent, then the parallel execution of the goals in the set is both correct and efficient. Finally, we show how to obtain a sufficient condition for the strict independence of a given set of goals. This is done first by considering the goals in isolation and then 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 a theoretical foundation for automatic parallelization tools.

3.1. Strict Goal Independence: Definition and Some Properties

Definition 2. (set of variables) Given any goal g, let us call $var(g)$ (respectively $var(g)$) the set of all the variables occurring in g. We also extend the definition to apply to terms, substitutions, etc.

Definition 3. (strict goal independence) Two goals g_1 and g_2 are said to be strictly independent for a given substitution θ iff $\text{var}(g_1\theta) \cap \text{var}(g_2\theta) = \emptyset$. A collection of goals is said to be strictly independent for a given θ iff they are pairwise strictly independent for θ. Also, a collection of goals is said to be strictly independent for a set of substitutions Θ iff they are strictly independent for any $\theta \in \Theta$. Finally, a collection of goals is said to be simply strictly independent if they are strictly independent for the set of all possible substitutions.

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. This same definition of strict independence can also be applied to terms without any change.

Example 2. Let us consider the two goals $p(x)$ and $q(y)$. Although they do not have any variable in common, they may not be 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. However, 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, the goals $p(a)$ and $q(b)$, where a and b are constants, are simply strictly independent (i.e., for any θ).

Note that if a term (or a goal) is ground, then it is strictly independent from any other term (or goal) – its set of variables is empty, so the intersection of this set with any other set of variables will be empty. Also, note that strict independence is symmetric, but not transitive: 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 with respect to 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 with respect to θ, because they share the variable w.

3.2. Strict Goal Independence is Sufficient for a Correct and Efficient Parallelization

Since strictly independent goals do not share variables, the renaming transformation of the parallel execution framework has no effect and can be simply ignored. Thus, the goals and their renamed versions coincide. I.e., following the same notation as in Section 2.2, $g_1\theta = g_1'$ and $g_2\theta = g_2'$.

Theorem 7. (strict independence and correct parallelization) Consider a substitution θ and two goals g_1 and g_2 which are strictly independent for θ. Let θ_1 be any answer substitution for $g_1\theta$, θ_2 any answer substitution for $g_2\theta\theta_1$, and thus $\theta_s = \theta\theta_1\theta_2$ the corresponding answer substitution for the sequential execution of $(g_1, g_2)\theta$. Also, let θ_2' be the corresponding answer substitution for $g_2\theta$, and $\theta_p = \theta\theta_1\theta_2'$ the corresponding answer substitution for the parallel execution of $g_1\theta$ and $g_2\theta$. Then $\theta_s \equiv \theta_p$.

Proof. we consider branches of the execution of g_1 and g_2 that result in success (for branches that result in failure the parallel and the sequential answers are trivially equivalent since they both fail). As argued before (Section 2.6) the corresponding branches in the parallel execution are also guaranteed to succeed and have the same number of steps. Regarding the resulting substitutions, by definition of strict independence $var(g_1\theta) \cap var(g_2\theta) = \emptyset$, and in general $domain(\theta_i) \subseteq var(g_i\theta) \cup NV$ for any i, where NV includes new variables introduced by the renaming steps in the execution of $g_i\theta$ which, by definition of the parallel framework are distinct from all other variables. Therefore $var(g_2\theta) \cap domain(\theta_1) = \emptyset$, which implies that $g_2\theta\theta_1 = g_2\theta$. Thus, independently of whether g_2 is pure or not, since an identical goal is being executed, then we may conclude that $\theta_2 \equiv \theta_2'$ and thus that $\theta_s \equiv \theta_p$.□

Thus, we have proved that it is correct to execute in parallel strictly independent goals, i.e., that their parallel execution (in the framework described in Section 2) returns the same computed answer substitution as their sequential execution. Note that this holds *both for pure and impure goals.* We will now show that parallel execution of strictly independent goals is also efficient.

Theorem 8. (strict independence and efficient parallelization) Consider two goals g_1 and g_2 and a substitution theta for which they are strictly independent. Let us call T_s the time for the sequential execution of $(g_1, g_2)\theta$, and T_p the time for their parallel execution. Then $T_p \leq T_s$.

Proof. It follows immediately from Theorem 6. In fact, if g_1 and g_2 are strictly independent for theta, then R is empty and there are no shared variables, so the condition of Theorem 1 (and thus that of Theorem 6) is trivially met.□

The results presented so far allow the parallel execution of any set of strictly independent goals in a resolvent while at worst preserving the efficiency of the sequential SLD resolution of this same resolvent. At the same time such results do not warrant the parallel execution of any set of goals which are not strictly independent. The implication is that such "dependent" goals should be executed using the usual left-to-right selection rule in order to maintain efficiency. Thus, the general rule is that for a given resolvent, any number of goals which are determined to be strictly independent can be started in parallel, but other goals cannot be started until the goals to their left on which they are dependent finish executing. However, we will show in a later section (Section 4) how the condition of strict independence can be relaxed to allow more goals to be executed in parallel.

In addition to the issue of the size of the search space itself, other efficiency issues have to be considered in the parallel execution of a set of strictly independent goals. One of them concerns how a particular set of strictly independent goals in the given resolvent is selected to be run in parallel. As shown in [15], the maximal independent set problem is NP-complete. In fact, the choice of the maximal set (i.e. running in parallel all the strictly independent goals that appear in the resolvent at a given point during the execution) is not even always the best one, because applying maximal parallelization at a given step may reduce the parallelism available at a later step (this applies even to Datalog programs [14]). Another important issue is the cost of determining goal independence at run-time. In the following sections we will show how the impact of these issues can be minimized by compile-time analysis. In fact, if both goal selection and independence checking are done completely at compile-time (i.e. only goals which can be determined to be independent at compile-time are run in parallel), then, using the results shown in this section, the parallel execution time can be guaranteed to be always shorter than (or in the worst case, equal to) the sequential one, since no additional time is spent in independence checking.

3.3. A Correct Local Condition for Strict Goal Independence

As mentioned before, checking the strict independence of a set of goals in the resolvent at run-time is straightforward, since it is sufficient to apply the definition. However, computing the set of variables for each goal and checking whether 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 refer to any such condition, which can in principle be any boolean function, as an "independence condition." We now formalize the notion of such a condition being "correct:"

Definition 4. (correct independence condition w.r.t. strict independence) An independence condition is said to be correct with respect to strict independence for a set of goals $g_1, \ldots, g_n$ and for a set of substitutions Θ iff for any substitution $\theta \in \Theta$ it holds that if $condition\theta$ is true then $g_1, \ldots, g_n$ are strictly independent for θ.

One particularly useful way of defining such a condition by using combinations of predefined predicates is as follows:

Definition 5. (*i_cond*) An *i_cond* is a special type of independence condition such that it is either "true" or a conjunction of one or more of the following tests:

- ground(x),
- indep(x,y).

where x and y can be goals, variables, or terms in general.

To understand the semantics of an *i_cond*, note that:

- $ground(x)$ is true when x is ground and false otherwise, and
- $indep(x, y)$ is true when x and y do not share variables and false otherwise, i.e. $indep(x, y)$ corresponds to a test for *goal-* and/or *term independence* as defined in Section 3.1. Note also that $indep(x, x)$ is true if and only if $ground(x)$ is.

For syntactic convenience, we extend an *i_cond* to also contain literals of the form

$$indep([(x_1, y_1), \ldots, (x_m, y_m)])$$

which is equivalent to

$$indep(x_1, y_1), \ldots, indep(x_m, y_m)$$

Example 3.

- $ground(x)$ is false for the substitutions $\theta_1 = \{x/f(y)\}$, $\theta_2 = \{x/y\}$, and $\theta_3 = \{x/f(g(1, y, 3))\}$ but is true for all of $\theta_4 = \{x/f(a)\}$, $\theta_5 = \{x/a\}$, and $\theta_6 = \{x/f(g(1, 2, 3))\}$;
- $indep(x, y)$ is false 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 is true for all of $\theta_4 = \{x/f(w),\ y/[1, 2, z]\}$, $\theta_5 = \{x/f(a)\}$, and $\theta_6 = \{x/f(y), y/a\}$;
- $ground([a, b, c])$ is always true, while $indep(f(x), g(y))$ is true for $\theta_1 = \{x/a\}$, but not for $\theta_2 = \{y/x\}$.

Given the primitives introduced above, a variety of *i_conds* can be constructed. 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 without making any assumptions regarding the current substitution, i.e. we have to assume that any substitution might be applied to the goals:

Definition 6. (local correctness of an independence condition w.r.t. strict independence) An independence condition is said to be locally correct with respect to strict independence for a set of goals $g_1, \ldots, g_n$ iff it is a correct independence condition with respect to strict independence for the set of all substitutions.

Clearly, the definition of independence is actually an *i_cond*, since it is the conjunction of all $indep(goal_i, goal_j)$ $\forall i, j, i \neq j$, and is locally correct by definition. However, as mentioned before, we now propose conditions that involve less run-time overhead.

Definition 7. (SVG,SVI) Given a collection of goals $g_1, \ldots, g_n$, let us define two sets SVG and SVI as follows:

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

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

$$ground(SVG) \wedge indep(SVI)$$

We will show that this *i_cond* is locally correct with respect to strict independence.

Theorem 9. The i_cond

$$ground(SVG) \wedge indep(SVI)$$

where SVG and SVI are computed on the collection of goals $g_1, \ldots, g_n$, is locally correct with respect to 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 $indep(SVI\theta)$ are true, then $var(g_1\theta) \cap var(g_2\theta) = \emptyset$. 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 will occur in a term resulting from 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 the first case assume that $v_1 \in var(g_1)$, $v_1 \in var(g_2)$. Then $v \in var(v_1\theta)$ is in contradiction with $ground(SVG\theta)$, since $ground(SVG\theta)$ is true iff $\forall v \in SVG$, $v\theta$ is ground.
- In the second case, suppose that $v_1 \in var(g_1)$ and $v_2 \in var(g_2)$. Then $v \in var(v_1\theta)$ and $v \in var(v_2\theta)$ where $(v_1, v_2) \in SVI$ is in contradiction with $indep(SVI)$, since $indep(SVI)$ is true iff $\forall v_1, v_2 \in SVI, var(v_1\theta) \cap var(v_2\theta) = \emptyset$.

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

In fact, it can also be easily shown that the above condition is also necessary for strict independence and thus equivalent to the definition.

For efficiency reasons, we can improve the conditions further by grouping pairs in SVI which share a variable x, such as $(x, y_1), \ldots, (x, y_n)$, by writing only one pair of the form $(x, [y_1, \ldots, y_n])$. By following on on this idea SVI can be defined in a more compact way as a set of pairs of sets as follows: $SVI = \{(V, W)$ such that $\exists i, j, i < j$, with $V = var(g_i) - SVG$ and $W = var(g_j) - SVG\}$. In some implementations this "compacted" set of pairs may be less expensive to check than that generated by the previous definition of SVI.

Example 4. The following table lists a series of sets of goals, their associated SVG and SVI sets, and a correct local *i_cond* with respect to strict independence:

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

As mentioned before, the main advantage in using $ground(SVG) \wedge indep(SVI)$ instead of the naive conjunction of $indep(g_i, g_j)$ is that the former can be more efficiently checked. More precisely, let us consider a reasonably efficient implementation of such checks and give an estimate of their cost:

- for $ground(x)$: traverse the entire structure of the term to which x is currently bound and check for the presence of any variable. Thus, the cost of such check is in this case proportional to the size (i.e. number of symbols) of x ($\mid x \mid$), i.e.
  - cost($ground(x)$) $\leq k \mid x \mid$, and
  - cost($ground(x)$) $= k \mid x \mid$ iff x is ground

  (where k is some constant).
- for $indep(x, y)$: traverse the term to which x is currently bound and bind all the variables to a fixed new constant; then traverse the term to which y is bound and see if that constant appears there.[1] Thus we have:
  - $k \mid x \mid \leq cost(indep(x, y)) \leq k(\mid x \mid + \mid y \mid)$, and
  - cost($indep(x, y)$) $= k(\mid x \mid + \mid y \mid)$ iff x and y are independent, and
  - cost($indep(x, y)$) $= k(\mid x \mid)$ iff x is ground.

It is easy to see that in general a groundness check is less expensive than an independence check, thus a solution where some independence checks are replaced by groundness checks is obviously preferable. It is also straightforward to show that the cost of an *i_cond* increases with the number of occurrences of variables in it. Thus, reducing the number of variable occurrences, as is done by the proposed *i_cond*, reduces cost. Finally, the number of variables in the *i_cond* generated can be used as a heuristic in compile-time estimation of test cost.

Example 5.

- Consider the simple collection of two goals

  $$p(x, y), q(y, z).$$

  The naive *i_cond* would be $indep(p(x, y), q(y, z))$, whose cost is, as described above, $\approx (\mid x \mid + \mid y \mid + \mid y \mid + \mid z \mid)$. In contrast, the *i_cond* that we would use is $ground(y), indep(x, z)$, whose cost is $\approx (\mid y \mid + \mid x \mid + \mid z \mid)$.
- Consider now the goals

  $$p(x, y, z), q(x).$$

  The cost of the naive *i_cond* ($indep(p(x, y, z), q(x))$) is $\approx (\mid x \mid + \mid y \mid + \mid z \mid + \mid x \mid)$, while the cost of ours (which is $ground(x)$) is only $\approx (\mid x \mid)$.

Conditions for the local correctness of an *i_cond* with respect to strict independence were first proposed, to the best of our knowledge, in [21]. 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 result in unnecessary checks).

[1] Note that the variables bound in the process should be unbound afterwards.

3.4. 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 Independent And Parallelism. As an example, in this section we will apply such results to a particular approach: independent/restricted and-parallelism. This approach combines compilation techniques 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 [21]. This model extends DeGroot's seminal work on *Restricted AND-Parallelism* [11] by providing backward execution semantics, improved graph expressions (&-Prolog's "Conditional Graph Expressions" –CGEs– and other related constructs),[2] and an efficient implementation model based on the Warren Abstract Machine (WAM) [33]. &-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, the CGE, is also provided. We now study the correctness of CGEs.

Definition 8. (CGE) A CGE (Conditional Graph Expression) is a structure of the form
(i_cond => $goal_1$ & $goal_2$ & ... & $goal_n$)
where 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 expression:

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

Therefore, the operational meaning of the CGE is

1. check i_cond,
2. if it succeeds execute the $goal_i, (i = 1, \ldots, n)$ in parallel, else execute them sequentially.

Since the $goal_i, (i = 1, \ldots, n)$ can themselves be CGEs, CGEs can be nested in order to create more complex execution graphs.

[2] &-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 conjunctive i_conds presented in this paper.

Definition 9. (local correctness of a CGE w.r.t. strict independence) A CGE
(i_cond => $goal_1$ & $goal_2$ & ... & $goal_n$)
is said to be locally correct with respect to strict independence iff i_cond is a correct local condition for $goal_1, \ldots, goal_n$ with respect to strict independence.

I.e., a CGE is locally correct with respect to strict independence if for any substitution θ, it holds that if $i\_cond\ \theta$ succeeds then $goal_1, \ldots, goal_n$ are strictly independent for θ. It follows directly from the results presented so far that a correct CGE with respect to strict independence can only generate correct and efficient parallelism (not taking into account the time involved in the checks, as well as scheduling and communication overheads):

Theorem 10. A CGE of the form
(**ground**(SVG), **indep**(SVI) => $goal_1$ & $goal_2$ & ... & $goal_n$)
where SVG and SVI are computed on the collection of goals $g_1, \ldots, g_n$, is locally correct with respect to strict independence for those goals.

PROOF. Follows immediately from Theorem 9 and Definition 9. □

Note that when CGEs are used to encode strict independence then it is guaranteed that no shared variables will ever appear at run-time among goals to be run in parallel. Therefore the renaming transformation is not ever necessary (and is not implemented in practice) in such a system.

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 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 [12, 23, 36, 32] for more details). A system which automatically performs such an annotation process and which also uses global information as described in section 3.5, is described in [36, 31, 18]. Some locally correct CGEs with respect to strict independence are shown in the following example.

Example 6. The following CGEs are locally correct with respect to strict independence:

```
( true                      => p & q(Y)            )
( 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,Z],W)  => p(X,Y,Z) & q(X,W)   )
```

3.5. A Correct Global Condition for Strict Goal Independence

In Section 3.3 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 coming from a clause, sometimes this condition may be too strong, i.e. it may be that simpler *i_cond*s than those presented in Section 3.3 are sufficient for guaranteeing the strict independence of the goals considered, for the substitutions affecting all the resolvents which contain these goals in any branch of the proof in which the clause is involved. 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, generally based on abstract interpretation [9], are applied to the program (e.g. see [10, 36, 29, 4, 31, 24]).

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 related to abstract interpretation. This will be done in a rather informal way, since a more thorough introduction to this technique is beyond the scope of the paper.

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 actual ("concrete") terms.[3] Elements from the abstract and concrete domain are often related by a "concretization" function γ, which given an element of the abstract domain returns the (possibly infinite) set of concrete elements that it represents. This switch from the Herbrand domain (i.e. the set of all concrete terms) to an abstract domain is used to approximate substitutions with abstract substitutions. An abstract substitution then represents a possibly infinite set of concrete substitutions.

Example 7. Consider the following abstract domain $\{free, any, ground\}$ (where $free$ represents the set of all free variables, $ground$ the set of all ground terms, and any the set of all terms), and the abstract substitution $\theta_\alpha = \{x/free, y/ground\}$. This substitution represents the set of all concrete substitutions $\theta \in \gamma(\theta_\alpha)$ such that x is bound to a free variable and y is bound to a ground term.

Definition 10. (entry mode or query form) An entry mode, or query form, E for a given program is a query such that its arguments are given in terms of an abstract domain. Its concretization, $\gamma(E)$ is the set of all queries obtained by replacing the

[3] Although a finite domain is mentioned in order to simplify the discussion note however that usefulness of the abstract interpretation technique is not necessarily limited to finite abstract domains.

arguments of E with elements of their concretization.

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

Example 8. The query form $p(ground, ground)$ represents all the possible queries of the form $p(t_1, t_2)$ where t_1 and t_2 are ground terms.

Definition 11. (global correctness of an *i_cond* w.r.t. strict independence) Let us consider a program P, a collection of goals $g_1, \ldots, g_n$ in the body of a clause of P and an entry mode E for P. Let us consider the resolution trees for any concrete query in $\gamma(E)$, and in those trees any node and the corresponding current substitution θ such that $g_1, \ldots, g_n$ appear leftmost in the resolvent corresponding to that node. Let us call Θ the set of all such θ. An *i_cond* is said to be globally correct with respect to the strict independence of $g_1, \ldots, g_n$ iff it is a correct independence condition for Θ.

In other words, we relax the local correctness of an *i_cond* with respect to 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. In practice, rather than executing the program for the (possibly infinite) set of all queries represented by E, an interpretation of the program over the abstract domain is performed and an abstract substitution Θ_α is computed which “approximates” the set Θ, where by approximation we mean set inclusion, i.e. we require that $\gamma(\Theta_\alpha) \supseteq \Theta$.

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 9. Consider the program
$p(x)$:- $q(x), r(x), s(x)$.
$q(a)$.
Suppose that we want to parallelize the execution of $r(x)$ and $s(x)$ in the first clause. Following the approach of Section 3.3, 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 Θ set for $r(x), s(x)$ given this query form is $\{\{x/a\}\}$ (and, in a possible abstract form $\{\{x/ground\}\}$), so it can be easily seen that the empty *i_cond*, which is not locally correct, is in contrast globally correct.

As we did for local correctness, we now construct an *i_cond* that is globally correct with respect to strict independence:

Definition 12. (SVG_g,SVI_g) Given a logic program P, a query mode, 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 $\forall\theta \in \Theta\ x\theta$ is ground.$\}$;
- $SVI_g = SVI - \{(x,y)$ such that $\forall\theta \in \Theta\ x\theta$ and $y\theta$ are strictly independent$\}$.

where the set Θ is as in Definition 11, or a safe approximation ($\gamma(\Theta_\alpha)$) of it.

Note that the resulting set of pairs defining SVI_g can also be compacted as described previously after defining SVI. Note also that since Θ represents the set of substitutions in all paths from a query form to the considered collection of goals, and since such a set may be infinite, the sets SVG_g and SVI_g may not be statically computable. On the other hand the techniques related to abstract interpretation mentioned above can be used to get an approximation of such sets in finite time.

Theorem 11. Given a logic program P, a query mode, 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 12, the i_cond

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

is globally correct for the strict independence of these goals.

PROOF. By theorem 9 we have that $(ground(SVG), indep(SVI))$ is correct for that set of goals for all substitutions. By definition we have that $(ground(SVG), indep(SVI)) = (ground(SVG_g), ground(SVG - SVG_g), indep(SVI_g), indep(SVI - SVI_g))$. In order to preserve correctness we have to make sure that if $(ground(SVG), indep(SVI))\theta$ is false then $(ground(SVG_g), indep(SVI_g))\theta$, is also false for all substitutions $\theta \in \Theta$. Let us assume that this does not hold. This can only happen if either $ground(SVG - SVG_g)\theta$ or $indep(SVG - SVI_g)\theta$ evaluate to false for any $\theta \in \Theta$, which would mean that there is at least one variable v in $SVG - SVG_g$ such that $v\theta$ is not ground or there is at least one pair of variables u, v in $SVI - SVI_g$ such that $u\theta$ and $v\theta$ are not independent. However, both those assumptions are in contradiction with the hypothesis regarding Θ. If an approximation $\gamma(\Theta_\alpha)$ of Θ is used instead, we have by definition of approximation that $\gamma(\Theta_\alpha) \supseteq \Theta$ and thus the hypothesis ensures that independence will hold for a larger set of substitutions than the actual ones that will occur and thus the condition is also globally correct. □

Example 10. (application - global correctness of CGEs w.r.t. strict independence) The CGE in the following clause is globally correct with respect to strict independence, given $\Theta = \{\theta_1, \theta_2\}$, $\theta_1 = \{x/f(a), y/a, z/w\}$, $\theta_2 = \{x/b, y/b, z/a, w/b\}$ (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 $SVG = \{x\}$, $SVI = \{(y,w),(z,w)\}$, $SVG_g = \emptyset$, and $SVI_g = \{(z,w)\}$. Also, note that this same CGE is not locally correct with respect to strict independence.

3.6. 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 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 contained in such clauses, and in others to simplify the *i_cond*.

Definition 13. (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.

Proposition 1. 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 leftmost occurrence of that variable in the clause body, 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 leftmost occurrence in any possible path. Therefore, the goals considered in the above proposition cannot be strictly independent because they share a variable.□

Proposition 2. 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 leftmost occurrence. Then the variables in F are (pairwise) strictly independent.

PROOF. consider two variables in F, say x and y. As above, since an existential variable cannot be bound before its leftmost occurrence for any possible path and for θ current substitution in that path $var(x\theta) = \{x\}$ and $var(y\theta) = \{y\}$, and thus $var(x\theta) \cap var(y\theta) = \emptyset$.□

This means that the independence condition for each pair of these variables can be deleted from the (locally correct) *i_cond*.

Except for cases such as those which will be treated in Section 4, 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 3.4.

Example 11. (application - existential variables and CGEs) The CGE in the following clause is globally correct with respect to 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 with respect to strict independence. Conversely note that the following CGE, although locally correct with respect to 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) ).
```

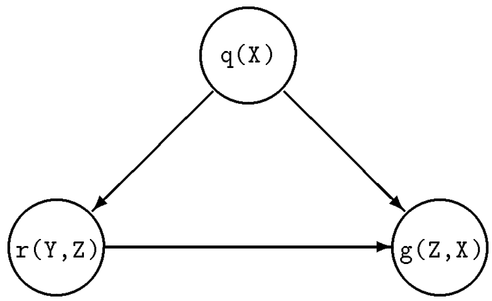
3.7. Application Example: Generation of Dependency Graphs

One way in which the dependencies between goals can be represented is in the form of a dependency graph [7, 23, 26, 27, 5, 32]. Informally, a dependency graph is a directed acyclic graph where each node represents a goal and each edge represents in some way the dependency between the connected goals. We will now show, using an example, how our approach subsumes such formalism and, therefore, that our results are sufficient for reasoning about its correctness.

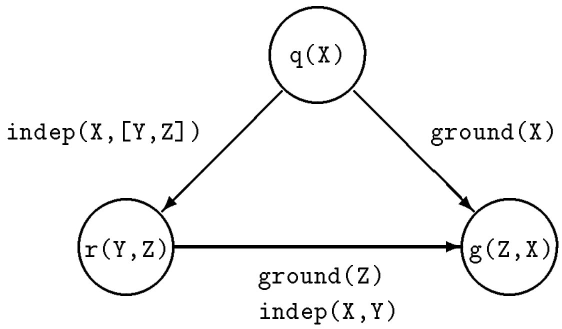
Consider the following clause:

`p(X,Y) :- q(X), r(Y,Z), g(Z,X).`

As mentioned before our efficiency results assume that the literal precedence relation given by the left-to-right selection rule, and that this precedence is preserved unless goals are determined to be independent. This precedence relation can be represented for the goals in the body of the clause above using a directed, acyclic graph as follows:

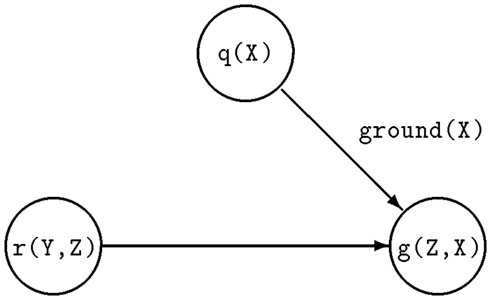


Using the rules described in Section 3.3, we can associate with each edge which connects a pair of literals the sufficient condition for their strict independence, thus resulting in the following dependency graph:



Note that while unlabeled edges state an unconditional precedence, edges labeled by a condition mean that the precedence between the two connected goals holds only if the condition is not satisfied by the (current) substitution.

In addition to the generation and proof of correctness of such graphs, our results from Sections 3.5 and 3.6 allow simplifications of the conditions. More concretely, in the clause under consideration `Z` is an existential variable with its first occurrence in `r`, thus `ground(Z)` can never be true, transforming the dependency between `r` and `g` into a hard one. Also, if we assume that it is known from global analysis that `Y` is always ground, then the edge from `q` to `r` can be eliminated since `Z`, having a first occurrence in `r`, is guaranteed to be free and thus independent from `X`, resulting in the following simplified graph:



This graph allows the parallel execution of `q` and `r`. Also, `g` has to be executed after `r`, and also after `q` if `X` is not ground when `q` starts executing. Such graphs can be encoded for example using Lin and Kumar's "bit-vector" approach [27] or compiled as &-Prolog expressions [32], i.e., for our example:

```
p(X,Y) :-
      ( ground(X) -> q(X) & ( r(Y,Z), g(Z,X) )
                  ;  q(X) & r(Y,Z),   g(Z,X)
      ).
```

Note that expressing more complex dependency graphs as &-Prolog expressions may require the use of &-Prolog's `wait` primitives [30].

4. NON-STRICT GOAL INDEPENDENCE

As mentioned before, our goal is to use the proposed framework for parallel independent execution, and to run in parallel as many goals as possible while maintaining correctness and efficiency with respect to the sequential execution. In the previous section we showed that strictly independent goals, i.e. goals which do not share any variable at run time, have these desirable properties. Here we will see how even some goals which do share variables can be run in parallel independently while being correct and efficient. Such goals will be called "non-strictly" independent. A particular case of such "non-strict independence" was hinted at by DeGroot in the Sor"qsort" example given in [11]. The MA3 system, presented in [36], incorporated an early concept of non-strict independence in its parallelization rules. Finally, the concept of "call instantiation correctness" was introduced by Winsborough and Waern in [38] which also allows a form of non-strict independence. Here, we follow the approach of [20] which generalizes these notions by proposing the concept of non-strict independence and then studying correctness and efficiency results for it, as well as proposing compile-time conditions. Furthermore, we propose a notion of non-strict independence which allows a slight relaxation of the conditions of [20].

4.1. Non-Strict Goal Independence: Definition

Definition 14. (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 15. (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 $G(v) = \{g_i\theta \mid v \in var(g_i\theta), v \in SH\}$. Let θ_i be any answer substitution for $g_i\theta$. The given collection of goals is non-strictly independent for θ if the following conditions are satisfied:

- $\forall v \in SH$, at most the rightmost $g \in G(v)$, say $g_j\theta$, nv-binds v in any θ_j;
- for any $g_i\theta$ (except the rightmost) containing more than one variable of SH, say $v_1, \ldots, v_k$, then $v_1\theta_i, \ldots, v_k\theta_i$ 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 (where that variable occurs) is not arbitrary. If the goal that nv-binds x is $g_i\theta$ and there is another goal $g_j\theta$, with $j > i$, that also contains x, then in the sequential execution with the usual left to right selection rule the execution of $g_i\theta$ may restrict the search space of $g_j\theta$, because θ_i may affect $g_j\theta$. In the parallel execution, $g_j\theta$ 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 creating aliases (of different shared variables) during the execution of one of the parallel goals which might affect goals to the right. In fact, an alias is also a restriction of the search space (to be avoided because of the reason discussed in the last paragraph) because it creates a dependence among different shared variables.

Example 12. Consider the collection of goals $(r(x,z,x), s(x,w,z), p(x,y), q(y))$ and an empty θ. Suppose 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. Furthermore, assume that after execution of $r(x,y,x)$ x and y are strictly independent and that after the execution of $s(x,w,z)$ x, w, z are strictly independent. Then, the original goals are non-strictly independent.

The conditions of Definition 15 above have been devised in order to ensure correctness and efficiency in general. However, if purity of goals is taken into account these conditions can be relaxed. Furthermore, the strict independence condition for answer substitutions can also be relaxed slightly in any case. Based on these ideas we propose the following concept of *generalized* non-strict independence:

Definition 16. (generalized 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 $G(v) = \{g_i\theta \mid v \in var(g_i\theta), v \in SH\}$. Let θ_i be any answer substitution for $g_i\theta$. The given collection of goals is non-strictly independent for θ if the following conditions are satisfied:

- $\forall x, y \in SH$, $\exists$ at most one $g_i\theta$ such that for any θ_i we have that $\{x,y\}\theta_i \not\equiv \{x,y\}$;
- $\forall x, y \in SH$, if $\exists\, g_i\theta$ meeting the condition above, then $\forall g_j\theta, j > i$, such that

$\{x, y\} \cap var(g_j\theta) \neq \emptyset$, g_j is a pure goal, and $\{x, y\}\theta_j \equiv \{x, y\}$ for all θ_j which are partial answers during the execution of $g_j\theta$.

Note that in the definition above the cases where $x = y$ are not excluded.

Intuitively, the first condition of the above definition requires that at most one goal modifies a shared variable or aliases a pair of variables. The second condition does not require that it be the rightmost goal containing the variables, but it does require that any goals to the right of the one modifying the variables be pure and do not "touch" such variables. This ensures that its search space could not have been pruned by any bindings made to those variables and therefore it is safe to run it in parallel, i.e. isolated from such bindings by the renaming transformation. Finally, note that, although left out in Definition 16 for simplicity, the notion could be generalized even further if "purity" is determined at the level of shared variables, rather than goals.

Example 13. Consider the collection of goals $p(x, y), q(x, y)$ in the resolvent, where:
p(a,y).
q(x,b).
then $p(x, y), q(x, y)$ are (generalized) non-strictly independent.

It is also interesting to notice that the conditions in Definitions 15 and 16 can be checked only through an analysis of the whole program and its possible executions, while the strict independence of a set of goals can always be checked by only looking at the goals and the current substitution. This will obviously be significant when we try to propose sufficient compile-time conditions for non-strict independence, since such conditions will necessarily have to involve global-level analysis. Thus, to use the same terminology as before, no "locally-correct" condition for non-strict independence will be proposed.

Proposition 3. If a collection of goals is strictly independent for a given θ, then it is also non-strictly independent for θ.

PROOF. In fact, the conditions in the definitions of non-strict independence are always satisfied for a collection of strictly independent goals, since strictly independent goals do not share any variable, and such conditions only need to be satisfied whenever shared variables are present. □

4.2. Non-Strict Goal Independence is Sufficient for a Correct and Efficient Parallelization

Since non-strictly independent goals may share variables, in general they will be renamed before their parallel execution, and then their renamed versions will be executed in parallel, followed by the execution of the back-binding goals. Thus, for correctness purposes, we need to compare (and prove the coincidence) of any answer substitution θ_s generated by the sequential execution of the selected goals, and that, θ_p, generated by the parallel execution of the renamed goals plus the sequential execution of the back-binding goals. We will do this for two goals. The generalization to n goals is straightforward. We will prove the result for the general case of Definition 16 in which the rightmost goal can be pure or impure, since, as shown in the following proposition it includes that of Definition 15.

Proposition 4. Non-strict independence implies generalized non-strict independence.

PROOF. It follows trivially from the fact that the conditions of Definition 15 imply those of Definition 16: the first condition of Definition 15 effectively prevents more than one goal from nv-binding shared variables (it only allows the rightmost containing them to do so), and the second one prevents any aliasing, so any pair of shared variables will be unchanged, modulo variable renaming, except for those which appear in the rightmost goal containing them (which is clearly a single goal). Furthermore, since the only goal allowed to perform any changes to shared variables is the rightmost containing them the second condition is trivially met (there are no goals to the right of it containing those variables). □

Theorem 12. (generalized non-strict independence and correct parallelization) Consider two goals g_1, g_2 in a resolvent $(g_1, g_2, g_3, \ldots, g_n)$ and a substitution θ. Let g_1 and g_2 be (generalized) non-strictly independent for θ (Definition 16). Let g_1', g_2', R be the new collection of goals obtained from the renaming transformation. Let θ_s be any answer substitution from the sequential execution of $(g_1, g_2)\theta$ and θ_p the corresponding answer substitution from the parallel execution of g_1' and g_2', followed by R. Then, $(g_3, \ldots, g_n)\theta_s \equiv (g_3, \ldots, g_n)\theta_p$.

PROOF. we study branches of the execution of g_1 and g_2 that result in success (for the others the parallel and the sequential are trivially equivalent since, as shown in Section 2.6, they both fail). The answers from the sequential execution of $(g_1, g_2, g_3, \ldots, g_n)\theta$ and those from $(g_1', g_2', R, g_3, \ldots, g_n)\theta$ are equivalent, since these two resolvents are equivalent, except for possible appearance of new infinite branches or different solutions in either g_1' or g_2'. By definition of the renaming transformation $g_1\theta = g_1'$ and thus their execution is identical. Thus, we only need to study g_2. We first consider the case in which g_2 is impure. Then by hypothesis for any $x, y \in SH$ we have that $\{x, y\}\theta_1 \equiv \{x, y\}$ and thus $g_2\theta\theta_1 \equiv g_2'$ and the execution of g_2' is equivalent to that of $g_2\theta\theta_1$ and thus the answers. On the other hand, if g_2 is pure, then, although it is possible that for some $x, y \in SH$ $\{x, y\}\theta_1 \not\equiv \{x, y\}$, by hypothesis $\{x, y\}\theta_i \equiv \{x, y\}$ is met for any partial answer θ_i of $g_2\theta$. Thus, the execution of $g_2\theta$ and $g_2\theta\theta_1$ are equivalent and thus that of $g_2\theta\theta_1$ and g_2'. Finally, since from the point of view of answers the sequential and parallel executions of (g_1', g_2') must be equivalent given that they have no variables in common the conclusion holds. □

That is, it is correct to execute (generalized) non-strictly independent goals in parallel. The next theorem shows that it is also efficient.

Theorem 13. (generalized non-strict independence and efficient parallelization) Consider two goals g_1 and g_2. Let us call T_s the time for the sequential execution of g_1 and g_2, and T_p the time for the parallel execution of g_1' and g_2'. Assume also that g_1 and g_2 are (generalized) non-strictly independent (Definition 16). Then $T_p \leq T_s$.

PROOF. It follows directly from Theorem 6 since either g_2 is pure and then the definition guarantees that the conditions of Theorem 1 hold (since $g_2\theta = g_2\theta\theta_1$) or g_2 is impure, in which case the definition guarantees that the conditions of Theorem 3 hold. □

Note that T_p is the time to execute in parallel g'_1 and g'_2, to which we must add the time to execute the k back-binding goals. Note also that, since the conditions of Theorem 2 are satisfied, back-binding goals (if executed), always succeed and thus represent a single, deterministic step. In practice, for a given implementation, this time can often be considered insignificant.

4.3. 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 (for example, 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 3.5 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 the θ_i as well (and, in the case of considering purity of goals, on their partial answers), which cannot be obtained in general from an *i_cond* check previous to the parallel execution of the goals (short of actually running the goals themselves). 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* with respect to non-strict independence.

Definition 17. (global correctness of an *i_cond* w.r.t. non-strict independence) An i_cond is said to be globally correct with respect to non-strict independence for a set of goals $g_1, \ldots, g_n$ in a program P and a set of substitutions Θ (defined as in Definition 11) iff, $\forall \theta \in \Theta$, if $i\_cond\theta$ is true, then $g_1, \ldots, g_n$ are non-strictly independent for θ.

Above, the set Θ is as defined in Definition 11. Also, in the following paragraphs, SVI and SVG are as defined in Section 3.3, and SVG_g and SVI_g are as in Section 3.5.

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 15 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.

As an example, 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 the terms to which such variables will be bound. 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 [31, 24]. 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 θ in Θ, which are all v-bound by all θ_i for all $g_i\theta$ in which they appear, except at most the rightmost one, and which are independent in θ_i from other variables in S appearing in the same goal. Intuitively, these are *program variables* which are bound to *run-time* variables for which the conditions in the definition of non-strict independence hold, and thus the conditions will only have to ensure that the rest of variables are also safe.

Given the set S, consider the set $SD = S \times S - \{(x,x), x \in S\} - \{(x,y), \exists g_i, x \in var(g_i), y \in var(g_i)\}$ (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 do not appear in the same goal$\}$.

Definition 18. (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 strictly independent due to global analysis.

In words, SVG_{ns} contains all SVG_g except those variables meeting the non-strict independence conditions. 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*:

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

(which, again, can be compacted as shown when defining SVI). The following theorem shows that this *i_cond* is sufficient for the non-strict independence of $g_1, \ldots, g_n$.

Theorem 14. The i_cond

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

where SVG_{ns} and SVI_{ns} are computed on the collection of goals $g_1, \ldots, g_n$ as in Definition 18 is globally correct with respect to 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 θ in Θ. 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 the terms to which the variables in SI are bound, 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 14. 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 Θ, that w and z as well as l and k are independent in Θ, and that x, y meet the single, rightmost goal nv-binding and non-aliasing conditions, we have the sets: $SVG_g = \{x, y, n, v\}$, $S = \{x, y\}$, $SVI_g = \{(z, k), (l, w), (l, k), (w, k)\}$, $SD = \emptyset$, $SI = \{z, w, k, l\}$, $SSI = \{(x, k), (y, w)\}$, $SIP = \{(w, z), (x, k)\}$. Thus, $SVG_{ns} = \{n, v\}$, and $SVI_{ns} = \{(w, k), (z, k), (l, w), (l, k), (y, w)\}$.

Because of the rather conservative way in which it is given, the global condition for non-strict independence provided can only be considered as a first approximation of what is possible to achieve in terms of compile-time detection of this type of independence. Furthermore, we have not treated the issue of the run-time renaming transformation, which, unlike in strict independence, cannot be avoided in general for non-strictly independent goals. It would be desirable to perform this transformation when possible at compile-time or, at least, to minimize the run-time work involved through compile-time analysis. As will be mentioned in the conclusions, these issues are left as future work.

Example 15. (application - global correctness of CGEs w.r.t. non-strict independence) Given the following goals in a difference-list quick-sort program
`qsort(S,,[P|Ls]), qsort(L,Ls,R)`
and the knowledge that `S`, `L`, and `P` are ground in Θ, that `Ls` in the first `qsort` goal is a leftmost occurrence (and therefore independent from all other variables), and that the first `qsort` call does not nv-bind `Ls`, the following CGE is globally correct with respect to non-strict independence:
`(indep(Sor,R) => qsort(S,Sor,[P|Ls]) & qsort(L,Ls,R))`.
Note that this is one of the cases hinted at above in which the actual renaming transformation can also be done at compile-time. Here `Ls` is known to be free before and after the execution of the first `qsort` call. Thus the program variable `Ls` coincides with the run-time variable that has to be renamed. Therefore, the following CGE is a correct compile-time encoding of the run-time renaming transformation (which can then be avoided):
`(indep(Sor,R) => qsort(S,Sor,[P|LsP]) & qsort(L,Ls,R)), LsP=Ls`.

Note that the dependency graph approach (mentioned in Section 3.7) could benefit in a straightforward way from the introduction of the concept of non-strict independence. This would allow the parallel execution of many more goals.

4.4. 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 $g_1, \ldots, g_n$ and let us suppose that some of the g_i are positive and some negative. Because of the above consideration (that can be formally derived also by appropriate global analysis), the following facts hold:

- if a shared variable x occurs only in negative goals or in at most one positive goal which is to the right of the negative ones, then the first condition of the definition of non-strict independence holds for x;
- if g_i is a negative goal then the second condition of the definition of non-strict independence holds for all the pairs of variables in this goal.

The above discussion is given in terms of run-time variables. However, it is also possible to exploit the presence of negative goals at compile time, as the following corollary shows:

Corollary 1. Given a collection of literals $g_1, \ldots, g_{n-1}, g_n$ in the body of a clause of a program, if g_i is a negative literal $\forall i = 1, \ldots, n-1$, then they are non-strictly independent.

PROOF. Consider any substitution θ. Then, for any shared variable v, at most $g_n\theta$ nv-binds v. Consider then any pair (x, y) of shared variables which appear in the same $g_i\theta, i \leq n-1$. Then, since $g_i\theta$ is negative, it will not nv-bind or alias them, so they will be strictly independent. Therefore, the two conditions for non-strict independence hold.□

Example 16. Consider the following clause:
$p(x, y, z, v, w)$:- $\neg r(x, v), \neg s(y, w), q(x, y), t(y, z)$.
For the first three literals the corollary holds and thus they are non-strictly independent. If all four literals are considered, then, it is straightforward to show that even if no global information is available, by combining concepts and conditions from strict and non-strict independence a globally correct *i_cond* for these four goals is simply `ground(y), indep(x,z)`.

Of course this discussion about negative literals assumes that the program has been written taking into account any problems that might occur when executing possibly non-ground negative goals.

5. USING STRICT AND NON-STRICT INDEPENDENCE IN PRACTICE

The concepts presented regarding strict and non-strict independence can be used in practice to obtain speedups with respect to the sequential execution. This has been shown for several benchmarks for example in [18]. As an example, in this section we present actual run-times from the result of parallelizing a medium-sized benchmark (`boyer`, a reduced version of the Boyer-Moore theorem prover, written by Evan Tick) which has the advantage of allowing the exploitation of both strict- and non-strictly independence, although to different degrees. This benchmark proves theorems in basically two steps: a rewriting step ("rewrite," which comprises most of the computation) and a tautology checking step ("prove"). Table 1 gives execution times for the benchmark running on the unoptimized version of the &-Prolog system [18], using 1-10 Sequent Balance processors, for the original,

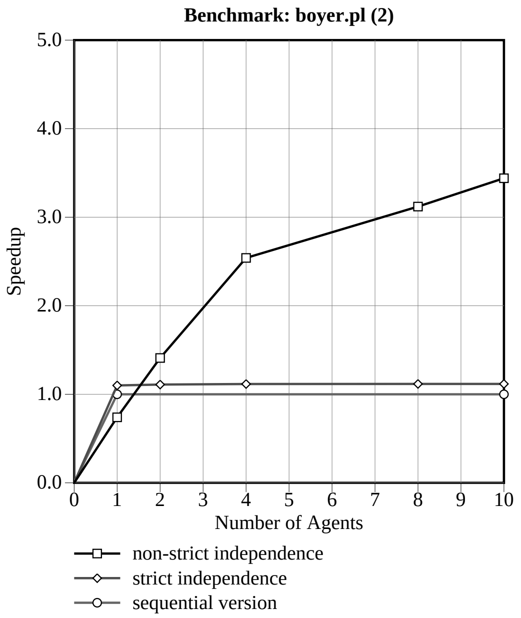


Figure 1. Speedup for `boyer(2)` - strict vs. non-strict independence

sequential program and for the cases in which the program has been parallelized using either strict- or non-strict independence. The results for the whole benchmark are represented in speedup form in Figure 1. It can be observed that while only a small amount of speedup can be obtained by using strict independence, reasonable speedups[4] can be obtained using the non-strict independence notion. It is interesting to observe, as shown in Figure 2, that strict-independence is relatively successful at parallelizing the "prove" part of the algorithm. On the other hand it is unsuccessful at parallelizing the "rewrite" part, while non-strict independence parallelizes both. The fact that, as can be seen in Table 1, "rewrite" represents the bulk of the computation, explains why, despite parallelizing the "prove" part correctly, no significant speedup is observed for strict independence in the whole benchmark. However, this result should not be taken as far as to imply that strict independence is not in general useful in practice. It has the advantage of being easier to detect than non-strict independence and, in fact, and as shown for example in [18], some programs can still be parallelized quite successfully using strict independence alone.

6. CONCLUSIONS AND FUTURE WORK

Much work has been done and is currently in progress in the compilation and implementation of independent and-parallelism in its various forms. In this paper we have provided a theoretical justification for such efforts, more general definitions

[4]This speedup can be made arbitrarily large by using appropriate data. In this case a theorem requiring a relatively small proof was used.

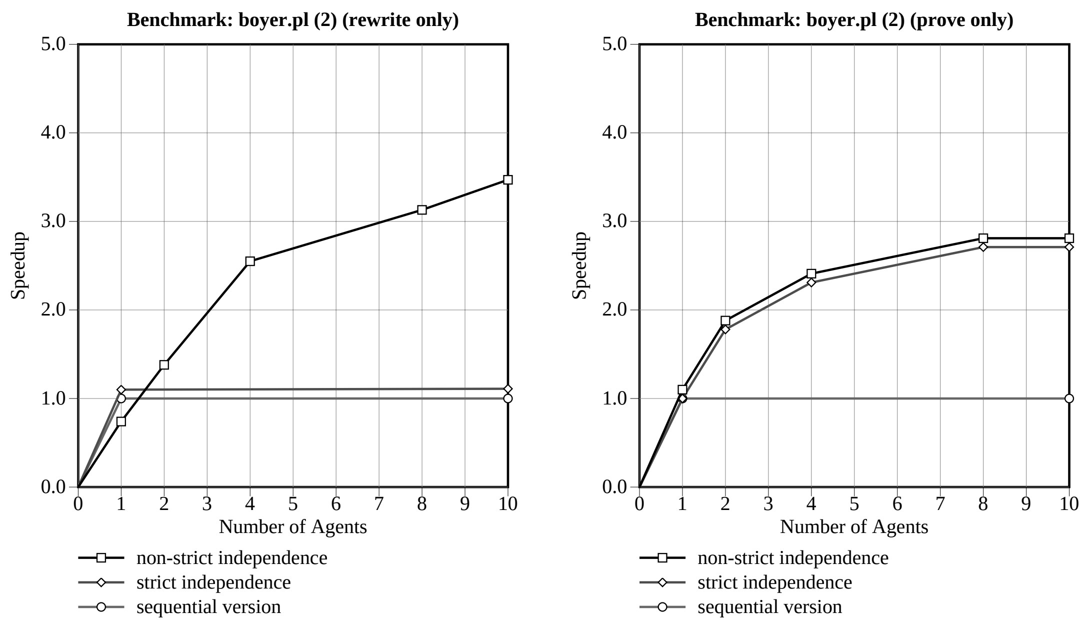


Figure 2. Speedup for "rewrite" and "prove" of `boyer(2)` - strict vs. non-strict independence

Nproc	seq	si	nsi
1	6338 (6179 + 159)	6338 (6179 + 159)	8479 (8320 + 159)
2	6339 (6179 + 160)	6269 (6179 + 90)	4479 (4389 + 90)
4	6339 (6179 + 160)	6238 (6169 + 69)	2488 (2419 + 69)
8	6339 (6179 + 160)	6228 (6169 + 59)	2029 (1970 + 59)
10	6339 (6179 + 160)	6228 (6169 + 59)	1838 (1779 + 59)

Table 1. Execution Time, `boyer.pl` (`rewrite` + `prove`) on Sequent Balance: 1-10 Processors, sequential vs. strict indep. vs. non-strict indep.

of independent and-parallelism which can extend their applicability, and a formal basis for the automatic exploitation of such kind of parallelism.

We have introduced a parallel execution framework and used it to reason about the correctness and efficiency of running goals in a resolvent in parallel independently. As a result of this we have identified two interesting classes of goals (one included in the other one) whose parallel execution is both correct and efficient. Goals in such classes are called respectively strictly and non-strictly independent.

More precisely, 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 execution time is reduced (or, in the worst case, it remains the same). We have then introduced the concept of non-strict independence and we have shown that the same results hold for non-strictly independent goals, 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 leftmost occurrence) are never strictly 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 the correctness and efficiency of user-provided annotations.

Because of its dependence on information to be obtained from global analysis, the exact nature of which is outside the scope of this paper, the compile-time conditions for non-strict independence given have been proposed only to serve as an example, and under quite simplistic assumptions regarding such information. This topic, which clearly needs to be developed further in view of the capabilities of particular analyzers, and the related one of determining when and how to perform the renaming transformation at compile-time, are proposed as future work.

Another subject for future study is the extension of the results of this paper to the constraint logic programming framework [25], which extends logic programming by replacing term equalities with constraints and the unification algorithm with any constraint solver. In fact, the "back-binding" goals and the conditions on them for ensuring correctness and efficiency also have a natural interpretation in terms of the constraint logic programming model as constraints that have to be satisfiable.

Finally, the model used in this paper has considered the parallel execution of entire proof trees associated with goals, since this reflects the operation of a significant class of models of and-parallel execution for non-deterministic logic languages. In other words, if two or more goals are found to be dependent, then their proof trees are explored one after the other (of course, parallelism is still allowed within the exploration of each one of the trees). However, we believe that the ideas and results presented in this paper are not inherently limited to this particular model and can be used in a quite straightforward manner also as a basis for reasoning about the correctness and efficiency of running in parallel parts of executions of goals smaller than a whole proof tree, down to a single resolution step. We will refer to these possibly smaller parts as "threads". Ultimately, all parallel execution is by nature independent at some level of granularity, and therefore much of what is conventionally referred to as "dependent and-parallelism" could also be considered as independent and-parallelism if the concept of independence is applied at the right level. The basis for the exploitation of the remaining dependent and-parallelism is the concept of determinism, which is with independence the other main principle governing parallel execution models because of its ability to also guarantee the "no slowdown" property. While the determinism principle allows safely running *deterministic* threads in parallel independently of whether they are dependent or not, the independence principles allow safely running *non-deterministic* threads in parallel, provided the independence conditions are met. Further exploration of these points of view is also proposed as future work.

ACKNOWLEDGMENTS

We thank Daniel Cabeza Gras, María J. García de la Banda, Pedro López García, Saumya Debray, Kevin Greene, Kalyan Muthukumar, Carlo Zaniolo, Kish Shen, Thomas Sjoland, and the anonymous (and not anonymous) referees for suggesting several improvements to our presentation. Francesca Rossi would also like to thank the Italian National Research Council for supporting her stay at MCC. Part of this work was performed in the context of the ESPRIT project 2471 "PEPMA" and CICYT project TIC90-1105-CE and has greatly benefited from discussions with other members of the partner institutions, most significantly from SICS and University of Bristol, as well as with members of the CLIP group at the Technical University of Madrid (UPM). The research described in this paper was done at MCC, University of Texas, Technical University of Madrid (UPM), and University of Pisa. The final version of this paper was produced under partial support from ESPRIT project 6707 "PARFORCE" and CICYT project TIC93-0976-CE.

REFERENCES

1. K. Apt and M. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–863, July 1982.
2. K.R. Apt. Introduction to logic programming (revised and extended version). Technical Report CS-R8826, Centre for Mathematics and Computer Science (CWI), Amsterdam, 1988. To appear as a chapter in *Handbook of Theoretical Computer Science*, North-Holland (J. van Leeuwen, editor).
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, Seattle,Washington, 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. S.-E. Chang and Y. P. Chiang. Restricted AND-Parallelism Execution Model with Side-Effects. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 350–368, 1989.
7. J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.
8. J. S. Conery. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors. In *Symp. on Logic Prog.*, pages 457–467, August 1987.
9. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Acm Symp. on Principles of Programming Languages*, pages 238–252, 1977.
10. S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.
11. D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.
12. D. DeGroot. A Technique for Compiling Execution Graph Expressions for Restricted AND-parallelism in Logic Programs. In *Proc. of the 1987 Int'l Supercomputing Conf.*, pages 80–89, Athens, 1987. Springer Verlag.
13. D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.
14. A. L. Delcher and S. Kasif. Some Results on the Complexity of Exploiting Dependency in Parallel Logic Programs. *The Journal of Logic Programming*, 6(3):229–241, May 1989.
15. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York, 1979.
16. G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.
17. S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46. MIT Press, June 1990.
18. M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.
19. M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.
20. M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.
21. 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.
22. 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.
23. D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming*, pages 284–297, 1988.
24. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.
25. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *ACM Symp. Principles of Programming Languages*, pages 111–119. ACM, 1987.
26. L. Kale. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Fourth IEEE Symposium on Logic Programming*, pages 125–133. IEEE, 1987.
27. 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.
28. J. W. Lloyd. *Logic Programming.* Springer-Verlag, 1987.
29. C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 463–475. Springer-Verlag, July 1986.
30. K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.
31. K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.
32. K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.
33. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.
34. D. H. D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.
35. D. H. D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.
36. 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*, pages 684–699, Seattle, Washington, August 1988. MIT Press.
37. H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.
38. 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, Seattle,Washington, 1988.