Independence and Search Space Preservation in Dynamically Scheduled Languages*

M. García de la Banda, M. Hermenegildo and K. Marriott

Abstract

This paper performs a further generalization of the notion of independence in constraint logic programs to the context of constraint logic programs with dynamic scheduling. The complexity of this new environment made necessary to first formally define the relationship between independence and search space preservation in the context of CLP languages. In particular, we show that search space preservation is, in the context of CLP languages, not only a sufficient but also a necessary condition for ensuring that both the intended solutions and the number of transitions performed do not change. These results are then extended to dynamically scheduled languages and used as the basis for the extension of the concepts of independence. We also propose several a priori sufficient conditions for independence and also give correctness and efficiency results for parallel execution of constraint logic programs based on the proposed notions of independence.

1 Introduction

Independence refers to the conditions that the run-time behavior of the goals to be run in parallel must satisfy in order to guarantee the correctness and efficiency of the parallelization with respect to the sequential execution. Correctness is guaranteed if the answers obtained during the parallel execution are equivalent to those obtained during the sequential execution. Efficiency is guaranteed if the no "slow-down" property holds, i.e., if the parallel execution time is guaranteed to be shorter or equal than the sequential execution time.

Previous work in the context of traditional logic programming languages [2, 6, 7, 8, 9] has concentrated on defining sufficient conditions which ensure the preservation of the search space of the goals to be run in parallel. The reasons for this are twofold. Firstly, efficiency was ensured by requiring that the amount of work performed for computing the answers during the parallel execution be equal to that performed in the sequential execution. Such work was measured in terms of the number of non failure transitions performed. Secondly, it was shown that if a goal g_1 cannot change the search space of a goal g_2 with respect to a given substitution θ, then the correctness and efficiency (measured in the above terms) of their parallel execution with respect to the sequential execution of $\langle g_1 : g_2, \theta \rangle$, is guaranteed.

Recently the concept of independence has been extended to the general context of the constraint logic programming (CLP) paradigm [11]. The work presented in [4] shows that a naive extrapolation of the LP definitions of independence to CLP is unsatisfactory (in fact, wrong) for two reasons. First, because interaction between variables through constraints is more complex than in the case of logic programming. Second, because the cost of executing a set of primitive constraints may depend on the order in which those primitive constraints are considered. Thus, optimizations which vary the intended execution order established by the user, such as parallel execution, can actually cause execution to slow-down.

In this paper we extend the notion of independence to dynamically scheduled languages, a class of second-generation logic programming languages that provide more flexible scheduling than those based on a

*This work was funded in part by ESPRIT project #7195 "ACCLAIM" and by CICYT project TIC93-0975-CE.

fixed scheduling rule. In this new class of languages computation generally proceeds following also some fixed scheduling rule but some calls are dynamically "delayed" until their arguments are sufficiently instantiated to allow the call to run efficiently. Such dynamic scheduling overcomes the problems associated with traditional languages and their fixed scheduling. First, it allows the same program to have many different and efficient operational semantics, as the operational behavior depends on which arguments are supplied in the query. Thus, programs behave efficiently as relations, rather than as functions. Second, the treatment of negation is sound, as negative calls are delayed until all arguments are ground. Third, it allows intelligent search in combinatorial constraint problems. Finally, dynamic scheduling allows a style of programming in which procedures are viewed as processes which communicate asynchronously through dependent variables.

The interest of such extension is threefold. Firstly, due to the benefits mentioned above, most real (constraint) logic languages already provide such flexible scheduling. Secondly, dynamic scheduling has a significant cost. These performance problems make those languages good candidates for many optimizations and, in particular, those based on the independence concept. Thirdly, dynamically scheduled languages have been considered as good target languages for the implementation of concurrent constraint logic languages [5, 15, 16]. The reasons for such decision include the semantic similarities and the progress on implementation and optimization techniques for dynamically scheduled languages and, in particular, the benefits provided by the work on, and implementations of, or- and and-parallel systems. However, taking advantage of such and-parallel systems will only be possible if the independence concept is extended to the class of logic languages with dynamic scheduling.

Unfortunately, the notions of independence developed so far are not valid for this class of languages which turned out to be too complex for a more or less straightforward extension, mainly due to the need to consider the effects of the awakening or delaying of some literals. This behavior makes necessary to consider the possible interleavings which appear in the presence of dynamic changes in the computation rule. The solution to this problem was found through a tighter formalization of the relationship between the concept of independence and the concept of search space preservation. We will first develop this new formalization for the CLP framework. The reason for this step is that this framework provides the tools for clearly defining such formal relationship. In particular, we will show that search space preservation is, in the context of CLP languages, not only a sufficient but also a necessary condition for ensuring that both the intended solutions and the number of transitions performed do not change. The results of this study not only serve as the basis for the extension to dynamically scheduled languages, but also significantly clarify the results on independence obtained in [4]. Finally the extension of the independence concept in the context of (constraint) dynamically scheduled languages is presented.

2 Background

In this section we present the notation which will be used throughout the paper. We will assume some background on the constraint logic programming scheme [11, 12].

Upper case letters generally denote collections of objects, while lower case letters generally denote individual objects. u, v, w, x, y, z will denote variables, t will denote a term, p, q will denote predicate symbols, f will denote a function symbol, a, h will denote atoms, P, Q will denote programs and g, G will denote goals. These symbols may be subscripted or have an over-tilde. $\tilde{x}$ denotes a sequence of distinct variables. $\exists_{-\tilde{x}}\phi$ denotes the existential closure of the formula ϕ except for the variables $\tilde{x}$. $\tilde{\exists}\phi$ denotes the full existential closure of the formula ϕ.

Let Σ and Var denote a set of function symbols and a denumerable set of variables, respectively. Let Π denote a set of predicate symbols such that $\Pi = \Pi_c \cup \Pi_P$ and $\Pi_c \cap \Pi_P = \emptyset$. A *primitive constraint* has the form $p(t_1, \cdots, t_n)$ where $t_1, \cdots, t_n$ are terms and $p \in \Pi_c$ is a predicate symbol. Every *constraint* is a conjunction of primitive constraints. The empty constraint is denoted ϵ. An *atom* has the form $p(t_1, \cdots, t_n)$ where $t_1, \cdots, t_n$ are terms and $p \in \Pi_P$. A *literal* is an atom or a primitive constraint, and will be usually denoted by b or l. A *CLP program* is a collection of clauses of the form $h \leftarrow B$ where h is an atom (the head) and B is a sequence $b_1, \cdots, b_n$ of literals (the body). We assume that the clauses are in *normalized* form, (or standard form), i.e., that all arguments in atoms are distinct variables, each variable occurring in at most one atom. A *goal* G (also denoted by g) is a sequence of literals. A *renaming* is a mapping from Var to Var. We let *Ren* be the set of renamings, and naturally extend renamings to mappings between atoms, clauses, and constraints.

Let $\mathcal{D}$ be a (Σ, Π_c)-structure and $\mathcal{L}$ be a class of (Σ, Π_c)-constraints. Constraint domains $(\mathcal{D},\mathcal{L})$ are expected to support the following tests and operations on constraints:

1. *Consistency* or *satisfiability* of a constraint c: $\mathcal{D} \models \tilde{\exists} c$.
2. *Implication* or *entailment* of a constraint c_1 by a constraint c_0: $\mathcal{D} \models c_0 \rightarrow c_1$.
3. *Projection* of a constraint c onto variables $\tilde{x}$: $\mathcal{D} \models \exists_{-\tilde{x}} c$.
4. Detection that, given a constraint c, there is only one value that a variable x can take that is consistent with c: $\mathcal{D} \models \exists z, \forall x, \tilde{y}\ c(x, \tilde{y}) \rightarrow x = z$. We say that x is *definite* in c and denote by $def(c)$ the set of definite variables in c.

We will particularize the general operational semantics described in [12] for CLP systems by assuming that the computation rule implies left-to-right execution order and that passive constraints are not allowed to appear. The motivation behind these assumptions is to start from a particular sequential semantics with which to establish the appropriate comparisons and avoid the problems posed by passive constraints which are subsumed by those appearing in dynamically scheduled languages, later considered.

The operational semantics is presented as a transition system on *states* $\langle G, c\rangle$ where G is a sequence of literals, and c is a constraint (called the *store*). There is one other state, denoted by *fail*. A *search rule* which selects (if necessary) a given clause of the program is assumed as given.

The transition system is also parameterized by a predicate $consistent(c)$ expresses a test for consistency of c. Usually it is defined by: $consistent(c)$ iff $\mathcal{D} \models \tilde{\exists} c$, that is a complete consistency test. However, systems may employ a conservative but incomplete test: if $\mathcal{D} \models \tilde{\exists} c$ then $consistent(c)$ holds but sometimes $consistent(c)$ holds although $\models \neg\tilde{\exists} c$.

Although we do not require the *consistent* function to be complete, it should satisfy the following two conditions. Firstly, it should not take variable names into account:

$$\text{Let } \rho \in Ren.\ consistent(c) \text{ iff } consistent(\rho(c))$$

Secondly, if a constraint is said to be consistent, all constraints entailed are also consistent:

$$\text{If } c \rightarrow c' \text{ and } consistent(c), \text{ then } consistent(c')$$

The transition rules in the modified operational semantics are:

- $\langle a : G, c\rangle \rightarrow_r \langle B :: G, c \wedge (a = h)\rangle$ if a is an atom, $r \equiv h \leftarrow B$ is a clause of program P renamed to new variables selected by the search rule, and h and a have the same predicate symbol[1].

- $\langle a : G, c\rangle \rightarrow_{rf} fail$ if a is an atom and, for every clause $r \equiv h \leftarrow B$ of P, h and a have different predicate symbols.

- $\langle c' : G, c\rangle \rightarrow_c \langle G, c \wedge c'\rangle$ if c' is a constraint and $consistent(c \wedge c')$ holds.

- $\langle c' : G, c\rangle \rightarrow_{cf} fail$ if c' is a constraint and $consistent(c \wedge c')$ does not hold.

Note that the conditions for applying each of the reduction rules are pairwise exclusive. This is necessary in order to simplify the definitions and theorems, and can always be achieved without loss of generality by grouping the application of some transition rules.

A *derivation* of a state s (called the *initial state* of the derivation) for a program P is a finite or infinite sequence of transitions $s_0 \rightarrow s_1 \rightarrow \cdots$, in which $s_0 \equiv s$. A state from which no transition can be performed

[1] Note that the conjunction with the store is always consistent since we are considering normalized clauses.

is a *final state*. A derivation is *successful* (also referred as a *refutation*) when it is finite and the final state has the form $\langle nil, \epsilon \rangle$. A derivation is *failed* when it is finite and the final state is *fail*. The constraint c is said to be a *partial answer* to state s if there is a derivation from s to a state with constraint c. An *answer* to state s is the last partial answer of a successful derivation.

The maximal derivations of a state can be organized into a *derivation tree* in which the root of the tree is the start state and the *children* of a node are the states the node can reduce to. The derivation tree represents the search space for finding all answers to a state and is unique up to variable renaming. Each branch of the derivation tree of state s is a derivation of s. Branches corresponding to successful derivations are called *success branches*, branches corresponding to infinite derivations are called *infinite branches*, and branches corresponding to failed derivations are called *failure branches*. We denote the set of answers to state s for program P by $answer_P(s)$, the partial answers by $partial_P(s)$, and the derivation tree by $tree_P(s)$.

3 Independence in Logic Programs

As mentioned in the introduction, in traditional logic languages the concept of independence has always been defined in terms of search space preservation. However, we argue that the reasons behind this relationship have never been completely clarified due to the complications posed by the composition of substitutions when formalizing the concepts. In this section we will first briefly reconstruct, in the LP context, the reasoning followed by the authors of [7, 8, 9] when establishing the connections between independence and preservation of search space. The sequential LP framework assumed is the traditional framework equipped with left-to-right computation rule. We will then formalize the concept of search space preservation and its relationship with independence, in the more general context of CLP. Such formalization not only provides the proof of correctness for the intuition that preserving search space preserves both the correctness and efficiency of the and-parallel execution in LP.

The independent and-parallelism model [2, 6, 10, 9] aims at independently (i.e., guaranteeing no need for communication, and possibly in different environments) running in parallel as many goals as possible while maintaining correctness and efficiency with respect to the sequential execution. Correctness and efficiency were respectively defined as requiring that the answers obtained from the parallel execution be equivalent to those obtained in their sequential execution, and that the no "slow-down" property hold. Efficiency was approximated by requiring that the amount of work performed for computing the answers during the parallel execution be equal to that performed in the sequential execution. In this context, independence was defined as the characteristics that the goals had to satisfy in order to ensure the correctness and efficiency of their parallel execution.

Assume that given the state $\langle g_1 : g_2 : G, \theta \rangle$ we want to execute g_1 and g_2 in parallel (the extension to a sequence of *consecutive* goals is straightforward). Then a possible execution scheme could be the following:

- execute $\langle g_1, \theta \rangle$ and $\langle g_2, \theta \rangle$ in parallel (in different environments) obtaining the answer substitutions θ_1 and θ_2 respectively,
- execute $\langle G, \theta_1\theta_2 \rangle$.

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

In this context, two main problems were detected. The first one, related to the *variable binding conflict* of [2], appears whenever during the parallel execution of $\langle g_1, \theta \rangle$ and $\langle g_2, \theta \rangle$ the same variable is bound to inconsistent values. Then, due to the definition of composition of substitutions [14, 1] the answers obtained by the parallel execution can be different than those obtained by the sequential execution, thus affecting the correctness of the model, as shown in [9].

Example 3.1 Consider the state $\langle \mathtt{p(x)} : \mathtt{q(x)}, \epsilon \rangle$ and the following program:

$\mathtt{p(x)} \quad \leftarrow \quad \mathtt{x = a}.$

$\mathtt{q(x)} \quad \leftarrow \quad \mathtt{x = b}.$

In this case, the sequential execution framework first executes $\langle \mathtt{p(x)}, \epsilon \rangle$, returning $\{\mathtt{x/a}\}$ and then executes $\langle \mathtt{q(x)}, \{\mathtt{x/a}\} \rangle$ which is reduced to the state $fail$. On the other hand, the parallel execution framework executes in parallel $\langle \mathtt{p(x)}, \epsilon \rangle$ and $\langle \mathtt{q(x)}, \epsilon \rangle$, returning $\{\mathtt{x/a}\}$ and $\{\mathtt{x/b}\}$, respectively. Then, the composition $\{\mathtt{x/a}\}\{\mathtt{x/b}\}$ results in the substitution $\{\mathtt{x/a}\}$. Thus we obtain a different answer. □

The second problem is due to the possibility of performing more work in the parallel execution than that performed during the sequential execution, thus affecting the efficiency of the model.

Example 3.2 Consider the state $\langle \mathtt{p(x)} : \mathtt{q(x)}, \epsilon \rangle$ and the following program:

$$\begin{array}{lll} \mathtt{p(x)} & \leftarrow & \mathtt{x = a}. \\ \mathtt{q(x)} & \leftarrow & \mathtt{x = b, proc, x = c}. \end{array}$$

where `proc` is very costly to execute.

While both the sequential and parallel execution will fail, their efficiency is quite different. While the sequential execution fails before executing `proc`, the parallel execution will first execute `proc` and then fail. □

A third problem was also detected whenever the goal to the left (g_1 in the above model) has no answers, since then the amount of work performed by the parallel execution may be greater than that performed by the sequential execution and, thus, the no slow-down property may not hold. However, this problem was solved by assuming that the processor executing such goal is able to kill the processors executing the goals to the right (g_2 above), and that such processor has a higher priority than those executing goals to the right. As a result, the problem of ensuring the correctness and efficiency of the independent and-parallelism model was focussed on ensuring that both the answers and the amount of work (measured in terms of number of non failure transitions in the derivation tree, in absence of the situation described above) obtained by the sequential and parallel execution of the goals, be the same.

The first solution proposed to ensure the two objectives was to only allow goals to be run in parallel if they do not share variables with respect to the current substitution [2]. This was formally defined in [7] as follows (and called "strict independence"):

Definition 3.1 [strict goal independence] Two goals g_1 and g_2 are said to be strictly independent with respect to a given substitution θ iff $\mathrm{vars}(g_1\theta) \cap \mathrm{vars}(g_2\theta) = \emptyset$. 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. ■

The authors of [7] proved that if goals g_1 and g_2 are strictly independent with respect to a given substitution θ, then the parallel execution of $\langle g_1, \theta \rangle$ and $\langle g_2, \theta \rangle$ obtains the same answers as those obtained by the sequential execution of $\langle g_1 : g_2, \theta \rangle$, and, in the absence of failure, parallel execution does not introduce any new work.

This sufficient condition is quite restrictive, since it can significantly limit the number of goals to be executed in parallel. However, as pointed out in [7], it has a very important characteristic: strict independence is an a priori condition (i.e., it can be tested at run-time before executing the goals).

Due to the restrictive nature of the notion of strict independence, there have been several attempts to identify a more general sufficient condition. The intuition behind such generalizations is that goals sharing variables could still be run in parallel when the bindings established for those shared variables satisfy certain characteristics. This was informally discussed in [6, 17, 18], formally defined in [7], refined in [8], and further refined in [9] as follows:

Definition 3.2 [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 3.3 [non-strict independence] Consider a collection of goals $g_1, \ldots, g_n$ and a 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 to $g_i\theta$. 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 instantiate a shared variable. 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.

At this point it was noticed that, due to the definition of the composition of substitutions, incorrect answers could be obtained even when there was no variable binding conflict for the shared variables.

Example 3.3 Consider the state $\langle \mathtt{p(x,y)} : \mathtt{q(y)}, \epsilon \rangle$ and the program:

$$
\begin{array}{lll}
\mathtt{p(x,y)} & \leftarrow & \mathtt{x = y.} \\
\mathtt{q(x)} & \leftarrow & \mathtt{x = a.}
\end{array}
$$

It is easy to check that `p(x,y)` and `q(y)` are non-strictly independent for ϵ. However, if we run $\langle \mathtt{p(x,y)}, \epsilon \rangle$ we obtain $\theta_p = \{\mathtt{x/z, y/z}\}$. If we now execute $\langle \mathtt{q(y)}, \theta_p \rangle$ we obtain the substitution $\theta = \{\mathtt{x/a, y/a, z/a}\}$. If, instead we execute $\langle \mathtt{q(y)}, \epsilon \rangle$ we obtain $\theta_q = \{\mathtt{y/a}\}$ thus ending with their composition $\theta_p\theta_q = \{\mathtt{x/z, y/z}\}$ as final substitution. This answer is obviously different from the θ obtained by the sequential execution, thus yielding an incorrect result. □

As noticed in both [8] and [9], this could be easily solved by defining a "parallel composition" which avoids these problems. Such parallel composition was defined in terms of "solving" the equations associated with the substitutions being composed. However, adopting a new definition of composition would have been required a revision of well known results in logic programming, which rely on the standard definition. As a result, the authors adopted a different solution which involved a renaming transformation. Informally, the renaming transformation of two goals g_1 and g_2 for a substitution θ, involves applying the substitution to both goals, eliminating any shared variables in the resulting goals 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 (for a formal definition see [9]).

Example 3.4 Consider the collection of goals ($r(x, z, x)$, $s(x, w, z)$, $p(x, y)$, $q(y)$) in a state (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:

$$r(x, z, x), s(x', w, z'), p(x'', y), q(y'), x = x', x = x'', y = y', z = z'. \square$$

Note that the first goal always remains unchanged. Goals of the form $x = x'$ above were called "back-binding" goals (denoted by BB) and are related to the back-unification goals defined in [13], and the closed environment concept of [3]. In this context, the parallel framework described above was redefined as follows:

Assume that given the state $\langle g_1 : g_2 : G, \theta \rangle$ we want to execute g_1 and g_2 in parallel (the extension to more than two goals is straightforward). Then, the execution scheme was defined as follows:

- apply the renaming transformation to $g_1\theta, g_2\theta$ obtaining g'_i, g'_j, BB,

- execute $\langle g_1', \epsilon\rangle$ and $\langle g_2', \epsilon\rangle$ in parallel (in different environments) obtaining the answer substitutions θ_1 and θ_2 respectively,
- execute $\langle BB, \theta_1\theta_2\rangle$ obtaining the answer substitution θ_3,
- execute $\langle G, \theta\theta_3\rangle$.

As before, it is assumed that the new variables introduced during the renaming steps in the parallel execution belong to disjoint sets.

Once the parallel framework was redefined, the notions of correctness and efficiency were also reconsidered. Correctness was not a significant problem since, in general, the answers provided by the parallel executions were the same (up to renaming) as the answers obtained in the sequential execution. Only an infinite derivation in the execution of $\langle g_2', \epsilon\rangle$ would yield a change, if there is no such infinite derivation in the sequential execution due to the effect of some answer to $\langle g_1, \theta\rangle$. However, since this was a particular case in which efficiency was also affected, the correctness problem was ignored in the knowledge that if efficiency was achieved this case could not happen and therefore correctness would also be ensured.

Inefficiency was then assumed to come from two sources. Firstly, due to a larger branch in the derivation tree associated with the parallel execution of $\langle g_2', \epsilon\rangle$, since such a tree would obviously imply more work. This was the point in which the notion of search space preservation was introduced. Unfortunately, such notion was never formally defined, the intuitive idea given for the preservation of the search space being the following: the search space of two states are the same if their associated derivation trees have the same "shape" [9]. This concept was later (in some sense erroneously) identified with the preservation of the number of non failure nodes in the respective derivation trees. Secondly, due to answers obtained during the parallel execution which when executing the back-bindings yield a failure, since this would again increase the work (backtracking, finding another answer, etc). Initially, concentrating on the success of the back-bindings introduced some confusion since it was easy to believe that if such bindings always succeed then the efficiency (and thus the correctness) of the parallel model was ensured. However, as pointed out in [9], this does not ensure the preservation of the amount of work in failed derivations.

Although the work developed in [9] does provide the basic results for LP, it is our thesis that the problems associated to the composition of substitutions, which yield the introduction of the renaming transformation, were one of the main reasons which prevented the clarification of all the issues introduced above. One the one hand the authors did not focus on the original variable binding conflict which, as we will show later, is the main source of the problems and provides the correct intuition for generalizing independence to the CLP paradigm. On the other hand they were just one step behind the definition of not only sufficient but also necessary conditions which ensure search space preservation and its relation with the success of back-bindings.

In the following section we will present such formalization in the context of CLP languages.

4 Independence and Search Space Preservation in CLP

We start from the observation that the view of composition of substitutions in CLP corresponds exactly with the "parallel composition" needed in [8]. What in the LP scheme would imply a reconsideration of the standard theory and results comes for granted in the CLP scheme. Therefore, we avoid the renaming transformation and redefine the parallel model as follows. Assume that given the state $\langle g_1 : g_2 : G, c\rangle$ we want to execute g_1 and g_2 in parallel (the extension to more than two goals is straightforward). Then the execution scheme is the following:

- execute $\langle g_1, c\rangle$ and $\langle g_2, c\rangle$ in parallel (in different environments) obtaining the answer constraints c_1 and c_r respectively,
- obtain c_s as the conjunction of $c_1 \wedge c_r$,

- execute $\langle G, c_s \rangle$.

As before, and in order to avoid problems when conjoining c_1 and c_r, it is assumed that the new variables introduced during the renaming steps in the parallel execution of the goals belong to disjoint sets.

In this context, the correctness of the parallel model requires that for each answer c_s obtained in the sequential execution there exist two answers c_1 to $\langle g_1, c \rangle$ and c_r to $\langle g_2, c \rangle$ which when conjuncted provide the same answer. Formally:

Definition 4.1 Let $\langle g_1 : g_2 : G, c \rangle$ be a state and P be a program. The parallel execution of g_1 and g_2 is correct iff for every $c_1 \in answers_P(\langle g_1, c \rangle)$ there exists a renaming $\rho \in Ren$ and a bijection which assigns to each $c_s \in answers_P(\langle g_2, c_1 \rangle)$[2] an answer $c_r \in answers_P(\langle g_2, c \rangle)$[3] with $c_s \leftrightarrow c_1 \wedge \rho(c_r)$. ■

The efficiency of the parallel model requires that, in absence of failure (i.e., when the goals to the left have at least one answer), the amount of work performed during the parallel execution be less or equal to that performed during the sequential execution. We will assume that the application of a particular transition rule has the same cost, independently of the state to which the transition is applied. Therefore, in order to obtain the amount of work performed in the execution of a given state s, we only need to know the cost of applying each particular transition rule and number of times in which each transition rule has been applied. Let TR be the set of different transition rules that can be applied. Let s be a state and $N(i, s)$ be the number of times in which a particular transition rule $i \in TR$ has been applied in $tree_P(s)$. Let $K(i)$ be the cost of applying a particular transition rule $i \in TR$, and assume that such cost is always greater than zero.

Definition 4.2 Let $\langle g_1 : g_2 : G, c \rangle$ be a state and P be a program. The parallel execution of g_1 and g_2 is efficient iff for every $c_1 \in answers_P(\langle g_1, c \rangle)$:

$$\sum_{i \in TR} K(i) * N(i, \langle g_2, c \rangle) \leq \sum_{i \in TR} K(i) * N(i, \langle g_2, c_1 \rangle) \quad ■$$

Note that the amount of work performed in conjoining the answers obtained from the parallel execution is not taken into account. We consider the cost of such operation as one of the overheads associated with the parallel execution (as creation of processors, scheduling, etc). Let us now focus on the definition of search space preservation and its relationship with the preservation of correctness and efficiency of the parallel execution with respect to the sequential execution. We assume that nodes in the derivation tree are labeled with their path. We say that two nodes n and n' in the derivation trees of states s and s', respectively, and with the same path *correspond* if either they are the roots of the tree (i.e., $n \equiv s$ and $n' \equiv s'$) or they have been obtained by applying the same reduction rule.

Definition 4.3 States s and s' have the same search space for program P iff there exists a (total) bijection which assigns to each node in $tree_P(s)$ its corresponding node in $tree_P(s')$. ■

The properties of the search space preservation and the particular characteristics of the two states, $\langle g_2, c \rangle$ and $\langle g_2, c_1 \rangle$, for which the search space preservation is required (the initial sequence of literals is the same and $c_1 \rightarrow c$) allow us to ensure the following result:

Theorem 4.4 Let $\langle g_1 : g_2 : G, c \rangle$ be a state and P a program. The parallel execution of g_1 and g_2 is correct if for every $c_1 \in answers_P(\langle g_1, c \rangle)$: the search spaces of $\langle g_2, c \rangle$ and $\langle g_2, c_1 \rangle$ are the same for P. ■

The proof comes directly from the following two lemmas.

---

[2]The suffix $_s$ will be associated to the arguments of the states obtained during the sequential execution.

[3]The suffix $_r$ will be associated to the arguments of the states obtained during the parallel execution of the goal to the right.

Lemma 4.5 Let $\langle g_2, c_1\rangle$ and $\langle g_2, c\rangle$ be two states with an identical sequence of literals, and P a program. There exists a renaming $\rho \in Ren$ such that for every two non failure nodes $s \equiv \langle G_s, c_s\rangle$ and $r \equiv \langle G_r, c_r\rangle$ with the same path in $tree_P(\langle g_2, c_1\rangle)$ and $\rho(tree_P(\langle g_2, c\rangle))$, respectively: $G_s \equiv G_r$. ■

Note that, as constructed, the domain of ρ is the range of ρ_2 and therefore $\rho(tree_P(\langle g_1, c_1\rangle)) \equiv tree_P(\langle g_1, c_1\rangle)$ and $\rho(\langle g_2, c\rangle) \equiv \langle g_2, c\rangle$. In the rest of this paper we assume that ρ satisfy such characteristic.

Lemma 4.6 Let $\langle g_2, c_1\rangle$ and $\langle g_2, c\rangle$ be two states with an identical sequence of literals such that $c_1 \rightarrow c$. Let P be a program and ρ be a renaming satisfying Lemma 4.5. Then, for every two nodes $s \equiv \langle G_s, c_s\rangle$ and $r \equiv \langle G_r, c_r\rangle$ with the same path in $tree_P(\langle g_2, c_1\rangle)$ and $\rho(tree_P(\langle g_2, c\rangle))$, respectively: $c_s \leftrightarrow c_1 \wedge c_r$. ■

Note that even if the *consistent* function associated with the underlying constraint system is not complete, and thus both c_s and $c_1 \wedge c_r$ possibly entail $fail$, the lemma is satisfied.

Those two lemmas, and the fact that search space preservation implies a bijection among answers, allow us to prove that search space preservation is sufficient for ensuring the correctness of the parallel execution. However, it is important to note that search space preservation is not necessary for ensuring correctness since it is possible that although nodes in successful derivations correspond, there exists at least two nodes in failure derivations which do not correspond.

Ensuring that efficiency is also guaranteed is even easier due to the definition of search space:

Theorem 4.7 Let $\langle g_1 : g_2 : G, c\rangle$ be a state, P be a program, and $c_1 \in answers_P(\langle g_1, c\rangle)$. If the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same for P, then $\sum_{i \in TR} K(i) * N(i, \langle g_2, c\rangle) = \sum_{i \in TR} K(i) * N(i, \langle g_2, c_1\rangle)$. ■

As before, note that search space preservation is not necessary for ensuring efficiency since, for example, if different transition rules have the same cost, even if the nodes in the trees do not correspond, the total cost can be the same. Then, we can state the following:

Theorem 4.8 Let $\langle g_1 : g_2 : G, c\rangle$ be a state and P a program. The parallel execution of g_1 and g_2 is efficient if for every $c_1 \in answers_P(\langle g_1, c\rangle)$: the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same for P. ■

Thus we have already shown that search space preservation is sufficient for ensuring the correctness and also for ensuring the efficiency of the parallel execution. However, we can go further and show that it is in fact necessary for ensuring that both correctness and efficiency hold. The following lemmas are instrumental for this result.

Lemma 4.9 Let $\langle g_2, c_1\rangle$ and $\langle g_2, c\rangle$ be two states with an identical sequence of literals such that $c_1 \rightarrow c$. Let P be a program. Then, for every two nodes s and r with the same path in $tree_P(\langle g_2, c_1\rangle)$ and $tree_P(\langle g_2, c\rangle$, respectively: s and r have been obtained with the same transition rule iff either $s \equiv r \equiv fail$ or they are both non failure nodes. ■

Lemma 4.10 Let $\langle g_2, c_1\rangle$ and $\langle g_2, c\rangle$ be two states with an identical sequence of literals such that $c_1 \rightarrow c$ and the search spaces of $\langle g_2, c_1\rangle$ and $\langle g_2, c\rangle$ are different for program P. Then, there exists a bijection which assigns to each node s in $tree_P(\langle g_2, c_1\rangle)$ for which there is no corresponding node in $tree_P(\langle g_2, c\rangle$, a node r in $tree_P(\langle g_2, c\rangle$ with the same path, such that s and r have been obtained applying the $\rightarrow_{cf}$ and $\rightarrow_c$ transition rule, respectively, and the parents of s and r correspond. ■

As a result, we can conclude that the only way in which the search spaces of $\langle g_2, c_1\rangle$ and $\langle g_2, c\rangle$, with $c_1 \rightarrow c$, can be different for a program P, is by pruning some branch of $tree_P(\langle g_2, c\rangle)$. Now we can state the following:

Theorem 4.11 Let $\langle g_1 : g_2 : G, c\rangle$ be a state and P a program. The parallel execution of g_1 and g_2 is correct and efficient iff for every $c_1 \in answers_P(\langle g_1, c\rangle)$ the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same for P. ■

Note that this implies that, in absence of failure, the amount of work performed during the parallel execution is in fact equal (never less) than that performed in the sequential execution, the speedup coming from the parallel execution of such work. From the results above we can also clarify two of the points mentioned in the summary of the results for LP. Let $\#nodes_P(s)$ be the number of non failure nodes in the derivation tree of state s for program P.

Corollary 4.12 Let $\langle g_2, c_1\rangle$ and $\langle g_2, c\rangle$ be two states with an identical sequence of literals such that $c_1 \rightarrow c$. Let P be a program. The search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same for P iff $\#nodes_P(\langle g_2, c\rangle) = \#nodes_P(\langle g_2, c_1\rangle)$. ■

This can explain why preservation of the search space and of the number of non failure nodes were identified. However, as we will see later, this identification cannot be done when dynamically scheduled languages are taken into account, since then a more constrained store c_1 can both prune and *enlarge* the search space.

The second confusing point, clarified in [9] for LP, was related to the success of the back bindings in the parallel framework based on the renaming transformation, can also be derived for CLP:

Corollary 4.13 Let $\langle g_1 : g_2 : G, c\rangle$ be a state, and P a program. If for every $c_1 \in answers_P(\langle g_1, c\rangle)$, the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same for P, the back bindings resulting from the parallel execution of g_1 and g_2 in the parallel model requiring the renaming transformation will always succeed. ■

Furthermore, inherited from Theorem 4.4, the success of the back bindings does not guarantee the preservation of the search space, thus confirming the results in [9].

At this point it is clear that, following the definitions of correctness and efficiency given above, search space preservation ensures both the correctness and efficiency of the parallel execution of independent goals. Furthermore, that search space preservation is not only a sufficient but also a necessary condition for ensuring both efficiency and correctness. However, there are still two issues related to the assumptions made when ensuring efficiency. Firstly, we have assumed that g_1 has at least one answer. If this is not true, the amount of work during the parallel execution might be increased. Such increment will depend on how the implemented system handles such situations. However, given the results above, if we assume the behavior of the system in case of failure proposed in [9], the same results can be obtained, thus ensuring efficiency also for those cases. Secondly, we have also assumed that the amount of work involved in applying a particular transition rule is independent of the state to which the rule is applied. Thus, there is one point which has not been taken into account, namely the changes in the amount of work involved when applying a particular transition rule to states with different constraint stores.

5 Operational Semantics for Dynamically Scheduled Languages

The operational semantics of a program P can be presented as a transition system on *states* $\langle G, c, D\rangle$ where G is a multi-set of literals, c is a constraint, and D is a multi-set of delayed literals formed by those literals playing a *passive* role.

The transition system is parameterized by four functions, namely *consistent*, $infer$, *delay* and *woken*. The functions $consistent(c)$ is that defined in Section 2. The function $infer(c, pc)$ takes an active constraint c and a passive constraint pc and computes a new active constraint c' and passive constraint pc'. It can be understood as relaxing pc in the presence of c to obtain more active constraints which are conjuncted to c to form c', pc being simplified to pc'. It is required that $\mathcal{D} \models (c \wedge pc) \leftrightarrow (c' \wedge pc')$ so that the information is neither lost nor guessed by $infer$. The function $delay(a, c)$ holds iff a call to atom a delays with the constraint c. The function $woken(D, c)$ returns the multi-set of atoms in the sequence of delayed literals D that are woken by constraint c. Note that the order of the calls returned by *woken* is system dependent. The transitions in the transition system are:

- $\langle G \cup a, c, D\rangle \rightarrow_d \langle G, c, D \cup a\rangle$ if a is an atom selected by the computation rule and $delay(a, c)$ holds.

- $\langle G \cup a, c, D\rangle \rightarrow_r \langle G \cup B, c, D \cup (a = h)\rangle$ if a is an atom selected by the computation rule, $delay(a, c)$ does not hold, $r \equiv h \leftarrow B$ is a clause of P renamed to new variables, and h and a have the same predicate symbol.
- $\langle G \cup a, c, D\rangle \rightarrow_{rf} fail$ if a is an atom selected by the computation rule, $delay(a, c)$ does not hold, and for every clause $r \equiv h \leftarrow B$ of P, h and a have different predicate symbols.
- $\langle G, c, D\rangle \rightarrow_w \langle G \cup D', c, D \setminus D'\rangle$ if $woken(D, c) = D'$, and D' is not empty.
- $\langle G \cup c', c, D\rangle \rightarrow_c \langle G, c, D \cup c'\rangle$ if c' is a constraint selected by the computation rule.
- $\langle G, c, D\rangle \rightarrow_i \langle G, c', (D \setminus s) \cup s'\rangle$ if $(c', s') = infer(c, s)$ and s is the set of constraints in D.
- $\langle G, c, D\rangle \rightarrow_t \langle G, c, D\rangle$ if $consistent(c)$.
- $\langle G, c, s\rangle \rightarrow_{tf} fail$ if $\neg consistent(c)$.

The notions of derivation, derivation tree, and final state can be obtained as straightforward extensions of those defined for the CLP context. $\langle c, D\rangle$ is said to be a *partial answer* to state s if there is a derivation from s to a state $\langle G, c, D\rangle$ and is said to be an *answer* if $\langle G, c, D\rangle$ is a final state and $G \equiv nil$. Given a finite derivation with final state $\langle nil, c, D\rangle$, the derivation is *successful* if D is empty, and it *flounders* otherwise. Thus, in this new context the answers are not always associated with successful derivations. As before, we denote the set of answers to state s for program P by $answer_P(s)$, the partial answers by $partial_P(s)$, and the derivation tree by $tree_P(s)$.

In order to simplify the definitions, in the rest of this paper we will not distinguish between the particular constraints returned by the $infer$ function and an equivalent result, i.e., if $infer(c_1, D_1) = (c^1, D^1)$, $infer(c_2, D_2) = (c^2, D^2)$, $c^1 \leftrightarrow c^2$ and $D^1 \leftrightarrow D^2$, we will say that $infer(c_1, D_1) = infer(c_2, D_2)$, $infer(c_1, D_1) = (c^2, D^2)$ and $infer(c_2, D_2) = (c^1, D^1)$.

6 First Case: Constraint Logic Programs with Passive Constraints

In this section we will discuss the problems posed by dynamically scheduled languages for the preservation of search space and the relationship between search space preservation and independence in this new context. In order to simplify the discussion we will consider three cases. In this section, we consider the case in which only constraints are allowed to be dynamically scheduled. In the next section, we will relax this condition by allowing atoms to be dynamically scheduled but we will only consider the problem of search space preservation in the case of states in which the sequence of delayed literals does not contain atoms. Finally, we will completely relax such conditions allowing any kind of delay behavior.

Since in this section the multi-set of delayed literals only contains constraints, by an abuse of notation, in the rest of this section we will consider it as the constraint formed by the conjunction of its elements. In order to start from a particular sequential semantics with which to establish the appropriate comparisons, we will particularize the general operational semantics previously described. Firstly, the computation rule implies left-to-right execution order. Thus, the element G of a state $s = \langle G, c, D\rangle$ will be represented by a *sequence* of literals instead of by a *multi-set*. Secondly, the operational semantics can be described by $\rightarrow_{rf}$, $\rightarrow_{rit}$, $\rightarrow_{cit}$, and $\rightarrow_{citf}$ transitions. We will consider such transitions as the *basic* transition rules. This allows us to consider transition rules which are pairwise exclusive, thus greatly simplifying the discussion. Note that, since we have assumed that only constraints are dynamically handled, transitions $\rightarrow_d$ and $\rightarrow_w$ are not allowed. Also, note that, given the basic transition rules proposed, for every state $\langle G, c, D\rangle$ in the derivation of another state s: $infer(c, D) = (c, D)$. We will extend this property to the queries, i.e., we will assume that for every state $\langle G, c, D\rangle$ considered: $infer(c, D) = (c, D)$.

Finally, we will assume that the conditions imposed in Section 2 for *consistent* hold and we will require similar conditions from $infer$. In particular, it should not take variable names into account:

$$\text{Let } \rho \in \mathit{Ren}.\ \rho(infer(c, D)) = infer(\rho(c), \rho(D))$$

It should be idempotent

$$\text{If } (c_1, D_1) = infer(c, D) \text{ then } (c_1, D_1) = infer(c_1, D_1)$$

and,

$$\text{If } c \leftrightarrow c_1 \wedge c_2 \text{ then } infer(c, D) = infer(c_1, D \wedge c_2)$$

6.1 Independence and Search Space Preservation

Let us redefine the and-parallel execution model in this new context. Assume that, given the state $\langle g_1 : g_2 : G, c, D\rangle$ in which D is a multi-set of constraints, and the program P in which no atoms are allowed to delay, we want to execute g_1 and g_2 in parallel (the extension to more than two goals is straightforward). Then the execution scheme is the following:

- execute $\langle g_1, c, D\rangle$ and $\langle g_2, c, D\rangle$ in parallel (in different environments) obtaining the answer constraints $\langle c_1, D_1\rangle$ and $\langle c_r, D_r\rangle$ respectively,
- obtain $(c_s, D_s) = infer(c_1 \wedge c_r, D_1 \wedge D_r)$
- execute $\langle G, c_s, D_s\rangle$.

As before, and in order to avoid problems when obtaining c_s and D_s, it is assumed that the new variables introduced during the renaming steps in the parallel execution of the goals belong to disjoint sets.

Let us now redefine correctness and efficiency of the model.

Definition 6.1 Let $\langle g_1 : g_2 : G, c, D\rangle$ be a state in which D is a multi-set of constraints and P be a program in which no atoms are allowed to delay. The parallel execution of g_1 and g_2 is correct iff for every $\langle c_1, D_1\rangle \in answers_P(\langle g_1, c, D\rangle)$ there exists a renaming $\rho \in \mathit{Ren}$ and a bijection which assigns to each $\langle c_s, D_s\rangle \in answers_P(\langle g_2, c_1, D_1\rangle)$ a $\langle c_r, D_r\rangle \in answers_P(\langle g_2, c, D\rangle)$[4] with $(c_s, D_s) = infer(c_1 \wedge \rho(c_r), D_1 \wedge \rho(D_r))$. ∎

As before, we assume that the cost of conjoining the answers, i.e., of obtaining (c_s, D_s), is negligible and that the application of a particular transition rule has the same cost (always greater than zero), independently of the state to which the transition is applied. Let TR be the set of different transition rules that can be applied. Let s be a state and $N(i, s)$ be the number of times in which a particular transition rule $i \in TR$ has been applied in $tree_P(s)$. Let $K(i)$ be the cost of applying a particular transition rule $i \in TR$.

Definition 6.2 Let $\langle g_1 : g_2 : G, c, D\rangle$ be a state in which D is a multi-set of constraints and P be a program in which no atoms are allowed to delay. The parallel execution of g_1 and g_2 is efficient iff for every $\langle c_1, D_1\rangle \in answers_P(\langle g_1, c, D\rangle)$: $\sum_{i \in TR} K(i) * N(i, \langle g_2, c, D\rangle) \leq \sum_{i \in TR} K(i) * N(i, \langle g_2, c_1, D_1\rangle)$. ∎

Since the definition of search space preservation is not affected by the changes in the operational semantics (and thus is the same as that given in Definition 4.3), let us focus on the properties of search space preservation and the relationship of search space preservation with the preservation of correctness and efficiency of the parallel execution with respect to the sequential execution.

The first point to be noticed is that a delayed (or passive) constraint can become active or remain delayed without affecting the search space of the subsequent branches in the derivation tree.

[4] As before, the suffixes $_s$ and $_r$ will be associated to the arguments of the states obtained during the sequential execution and the parallel execution of the goal to the right, respectively.

Example 6.1 Consider the state $\langle \mathtt{p(x,y)} : \mathtt{q(y)}, \epsilon, \mathtt{x * w = v} \rangle$ and the program:

$$
\begin{array}{lll}
\mathtt{p(x,y)} & \leftarrow & \mathtt{x = y.} \\
\mathtt{q(y)} & \leftarrow & \mathtt{y = 4, s(y,z).} \\
\mathtt{s(y,z)} & \leftarrow & \mathtt{z > y.} \\
\mathtt{s(y,z)} & \leftarrow & \mathtt{z < y.}
\end{array}
$$

where no atom can be delayed. The only answer obtained for $\langle \mathtt{p(x,y)}, \epsilon, \mathtt{x * w = v} \rangle$ is $\langle \mathtt{x = y}, \mathtt{x * w = v} \rangle$. Figure 1 shows the derivation tree for states $\langle \mathtt{q(y)}, \mathtt{x = y}, \mathtt{x * w = v} \rangle$ and $\langle \mathtt{q(y)}, \epsilon, \mathtt{x * w = v} \rangle$. Note that the renaming steps have been avoided for simplicity. It is clear that although the delayed constraint `y*x=w` becomes active at different points of the execution, the derivation trees for $\langle \mathtt{q(y)}, \mathtt{x = y}, \mathtt{x * w = v} \rangle$ and $\langle \mathtt{q(y)}, \epsilon, \mathtt{x * w = v} \rangle$, are the same .□

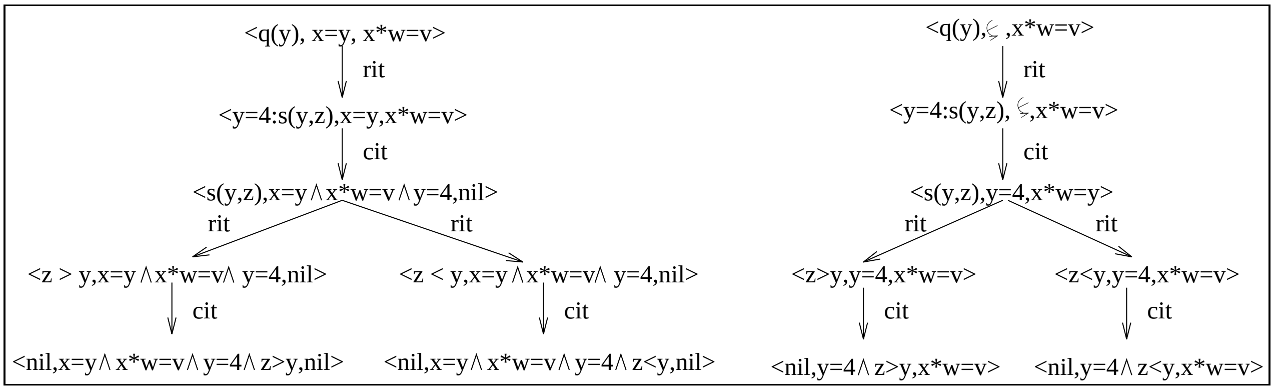

Figure 1:

Example 6.2 Consider the state $\langle \mathtt{p(x)} : \mathtt{q(x,y)}, \epsilon, nil \rangle$ and the program:

$$
\begin{array}{lll}
\mathtt{p(x)} & \leftarrow & \mathtt{x = 4.} \\
\mathtt{q(x,y)} & \leftarrow & \mathtt{y * x = w, s(x,w).} \\
\mathtt{s(x,w)} & \leftarrow & \mathtt{x > w.} \\
\mathtt{s(x,w)} & \leftarrow & \mathtt{x < w.}
\end{array}
$$

where no atom can be delayed. The only answer obtained for $\langle \mathtt{p(x,y)}, \epsilon, \mathtt{nil} \rangle$ is $\langle \mathtt{x = 4}, nil \rangle$. It is easy to see that although the constraint `y*x=w` becomes delayed in both branches of the the derivation tree for state $\langle \mathtt{q(y)}, \epsilon, \mathtt{nil} \rangle$ and does not become delayed in any branch of the the derivation tree for state $\langle \mathtt{q(y)}, \mathtt{x = 4}, \mathtt{nil} \rangle$, the derivation trees of those two states are the same.□

The situation is similar to that studied in Section 4 since, again, the search space can only be affected by pruning some branches. Therefore, we can still state that search space preservation is sufficient for ensuring correctness:

Theorem 6.3 Let $\langle g_1 : g_2 : G, c, D \rangle$ be a state in which D is a multi-set of constraints and P be a program in which no atom can be delayed. The parallel execution of g_1 and g_2 is correct if for every $\langle c_1, D_1 \rangle \in answers_P(\langle g_1, c, D \rangle)$, the search spaces of $\langle g_2, c, D \rangle$ and $\langle g_2, c_1, D_1 \rangle$ are the same for P. ■

The proof comes from the following two lemmas, which are extensions of the Lemma 4.5 and Lemma 4.6, respectively.

Lemma 6.4 Let $\langle g_2, c_1, D_1\rangle$ and $\langle g_2, c, D\rangle$ be two states with an identical sequence of literals in which both D_1 and D are multi-sets of literals, and P be a program in which no atom can delay. There exists a renaming $\rho \in Ren$ such that for every two non failure nodes $s \equiv \langle G_s, c_s, D_s\rangle$ and $r \equiv \langle G_r, c_r, D_r\rangle$ with the same path in $tree_P(\langle g_2, c_1, D_1\rangle)$ and $\rho(tree_P(\langle g_2, c, D\rangle)$, respectively: $G_s \equiv G_r$. ■

Given this result, we can ensure that, as constructed, $\rho(tree_P(\langle g_1, c_1, D_1\rangle)) \equiv tree_P(\langle g_1, c_1, D_1\rangle)$ and $\rho(\langle g_2, c, D\rangle) \equiv \langle g_2, c, D\rangle$. In the rest of this section we assume that ρ satisfies such conditions.

Lemma 6.5 Let $\langle g_2, c_1, D_1\rangle$ and $\langle g_2, c, D\rangle$ be two states with an identical sequence of literals in which both D and D_1 are multi-sets of constraints, $c_1 \rightarrow c$ and $c_1 \wedge D_1 \rightarrow D$. Let P be a program in which no atom is allowed to delay and ρ be the renaming from Lemma 6.4. Then, for every two nodes $s \equiv \langle G_s, c_s, D_s\rangle$ and $r \equiv \langle G_r, c_r, D_r\rangle$ with the same path in $tree_P(\langle g_2, c_1, D_1\rangle)$ and $\rho(tree_P(\langle g_2, c, D\rangle)$, respectively: $(c_s, D_s) = infer(c_1 \wedge c_r, D_1 \wedge D_r)$. ■

Note that even if the *consistent* function associated with the underlying constraint system is not complete, and thus c_s possibly entails $fail$, the theorem is satisfied.

Those two lemmas, and the fact that search space preservation implies a bijection among answers, allow us to prove that search space preservation is sufficient for ensuring the correctness of the parallel execution. Ensuring that the efficiency is also guaranteed is even easier due to the definition of search space.

Theorem 6.6 Let $\langle g_1 : g_2 : G, c, D\rangle$ be a state in which D is a multi-set of constraints, P a program in which no atom is allowed to delay, and $\langle c_1, D_1\rangle \in answers_P(\langle g_1, c, D\rangle)$. If the search spaces of $\langle g_2, c, D\rangle$ and $\langle g_2, c_1, D_1\rangle$ are the same for P, then:

$$\sum_{i \in TR} K(i) * N(i, \langle g_2, c, D\rangle) = \sum_{i \in TR} K(i) * N(i, \langle g_2, c_1, D_1\rangle). \quad ■$$

Thus we can ensure the following:

Theorem 6.7 Let $\langle g_1 : g_2 : G, c, D\rangle$ be a state in which D is a multi-set of constraints, and P be a program in which no atoms are allowed to delay. The parallel execution of g_1 and g_2 is efficient if for every $\langle c_1, D_1\rangle \in answers_P(\langle g_1, c, D\rangle)$ the search spaces of $\langle g_2, c, D\rangle$ and $\langle g_2, c_1, D_1\rangle$ are the same for P. ■

We have now shown that search space preservation is sufficient for ensuring the correctness and also for ensuring the efficiency of the parallel execution. However, analogously to the case in which no delayed constraints were allowed, we can go further and show that it is in fact necessary for having both correctness and efficiency. The following lemmas, extensions of Lemma 4.9 and Lemma 4.10 respectively, are instrumental for this result.

Lemma 6.8 Let $\langle g_2, c_1, D_1\rangle$ and $\langle g_2, c, D\rangle$ be two states with an identical sequence of literals in which both D and D_1 are multi-set of constraints, $c_1 \rightarrow c$ and $c_1 \wedge D_1 \rightarrow D$. Let P be a program in which no atom is allowed to delay. For every two nodes s and r with the same path in $tree_P(\langle g_2, c_1, D_1\rangle)$ and $tree_P(\langle g_2, c, D\rangle$, respectively: s and r have been obtained with the same transition rule iff either $s \equiv r \equiv fail$ or they are both non failure nodes. ■

Lemma 6.9 Let P be a program in which no atom is allowed to delay. Let $\langle g_2, c_1, D_1\rangle$ and $\langle g_2, c, D\rangle$ be two states with an identical sequence of literals in which both D and D_1 are multi-set of constraints, $c_1 \rightarrow c$, $c_1 \wedge D_1 \rightarrow D$ and the search spaces of $\langle g_2, c_1, D_1\rangle$ and $\langle g_2, c, D\rangle$ are different for P. Then, there exists a bijection which assigns to each node s in $tree_P(\langle g_2, c_1, D_1\rangle)$ for which there is no corresponding node in $tree_P(\langle g_2, c, D\rangle$, a node r in $tree_P(\langle g_2, c, D\rangle$ with the same path, such that s and r have been obtained applying the $\rightarrow_{citf}$ and $\rightarrow_{cit}$ transition rule, respectively, and the parents of s and r correspond. ■

As a result we can state the following:

Theorem 6.10 Let $\langle g_1 : g_2 : G, c, D\rangle$ be a state in which D is a multi-set of constraints, and P be a program in which no atoms are allowed to delay. The parallel execution of g_1 and g_2 is correct and efficient iff for every $\langle c_1, D_1\rangle \in answers_P(\langle g_1, c\rangle)$ the search spaces of $\langle g_2, c, D\rangle$ and $\langle g_2, c_1, D_1\rangle$ are the same for P. ■

Those results allow us to extend in a straightforward way the corollaries obtained regarding the preservation of the number of non failure nodes and the success of the back-bindings.

Therefore, following the definitions of correctness and efficiency given above, we can ensure that in the restricted context considered, search space preservation ensures both the correctness and efficiency of the parallel execution of goals. Furthermore, that search space preservation is not only a sufficient but also a necessary condition for ensuring that both efficiency and correctness hold.

We now extend the different notions of independence presented in [4] to this new context, based on the previous results.

6.2 Weak Independence

Weak independence aims at characterizing those sets of goals for which once there are no answers for at least one goal in the set, execution can safely backtrack to the choice-point placed just before the left-most goal, skipping all the choice-points in between. Thus, we are just interested in a characterization in which given a collection of goals $g_1 : \cdots : g_n$, constraint c, multi-set of constraints D and program P, if there exists $\langle c_1, D_1\rangle \in answers_P(\langle g_1 : \cdots : g_{i-1}, c, D\rangle)$ with $answers_P(\langle g_i, c_1, D_1\rangle) = \emptyset$, then, for every $\langle c_2, D_2\rangle \in answers_P(\langle g_1 : \cdots : g_{i-1}, c, D\rangle) : answers_P(\langle g_i, c_2, D_2\rangle) = \emptyset$. This characterization is related to the preservation of search space in the finite, non failure branches of the derivation tree, and thus can be defined as follows:

Definition 6.11 [weak independence] Goals g_1 and g_2 are weakly independent for constraint c, multi-set of delayed constraints D, and program P in which no atoms can be delayed iff

$$\forall\langle c_1, D_1\rangle \in answer_P(\langle g_1, c, D\rangle) \text{ and } \forall\langle c_r, D_r\rangle \in answer_P(\langle g_2, c, D\rangle) : consistent(c_s)$$

where c_s is the active constraint obtained by $infer(c_1 \wedge c_r, D_1 \wedge D_r)$. A collection of goals $g_1 : \cdot : g_n$ is weakly independent for a given c, D and P iff for every goal $g_i, 1 \leq i \leq n$: g_i and the goal $g_1 : \cdots : g_{i-1}$ are weakly independent for c, D and P. Also, a collection of goals is weakly independent for a set of constraints (interpreted as their disjunction) C, D, and program P iff they are weakly independent for any $c \in C$, D, and P. Finally, a collection of goals is simply weakly independent for D and P iff they are weakly independent for the set of all possible constraints, D, and P. ■

Given the following result:

Lemma 6.12 Goals g_1 and g_2 are weakly independent for constraint c, multi-set of constraints D, and program P in which no atom is allowed to delay iff $\forall\langle c_1, D_1\rangle \in answer_P(\langle g_1, c, D\rangle)$: there exists a bijection which assigns to each node in a successful branch of $tree_P(\langle g_2, c, D\rangle)$ a corresponding node in a successful branch of $tree_P(\langle g_2, c_1, D_1\rangle)$. ■

we can ensure that:

Theorem 6.13 Let $g_1 : \cdots : g_n$ be a collection of weakly independent goals for constraint c, multi-set of constraints D and program P. Let $g_i, 1 \leq i \leq n$ be a goal such that there exists $c_1 \in answers_P(\langle g_1 : \cdots : g_{i-1}, c, D\rangle)$ with $answers_P(\langle g_i, c_1, D_1\rangle) = \emptyset$. Then, for every $c_2 \in answers_P(\langle g_1 : \cdots : g_{i-1}, c, D\rangle) : answers_P(\langle g_i, c_2, D_2\rangle) = \emptyset$. ■

6.3 Strong Independence

Strong independence is aimed at detecting goals whose parallelization, when executed in different environments, is guaranteed to be correct and efficient. Thus the definition we are looking for is the following:

Definition 6.14 [strong independence] Goal g_2 is strongly independent of goal g_1 for constraint c, multi-set of delayed constraints D, and program P in which no atoms can be delayed iff

$$\forall \langle c_1, D_1 \rangle \in answer_P(\langle g_1, c, D \rangle) \text{ and } \forall \langle c_r, D_r \rangle \in partial_P(\langle g_2, c \rangle) : consistent(c_s)$$

where c_s is the active constraint obtained by $infer(c_1 \wedge c_r, D_1 \wedge D_r)$. A collection of goals $g_1 : \cdot : g_n$ is strongly independent for a given c, D, and P iff for every $g_i, 1 \leq i \leq n$: g_i is strongly independent of the goal $g_1 : \cdots : g_{i-1}$ for c, D, and P. Also, a collection of goals is strongly independent for a set of constraints (interpreted as their disjunction) C, D,and P iff they are strongly independent for any $c \in C$, D, and P. Finally, a collection of goals is simply strongly independent for D and P iff they are strongly independent for the set of all possible constraints and D and P. ■

Then, analogously to the properties shown by [4], we can show that:

Theorem 6.15 Goal g_2 is strongly independent of goal g_1 for constraint c, multi-set of delayed constraints D, and program P in which no atoms can be delayed iff $\forall \langle c_1, D_1 \rangle \in answers_P(\langle g_1, c, D \rangle)$:

$$\text{the search spaces of } \langle g_2, c, D \rangle \text{ and } \langle g_2, c_1, D_1 \rangle \text{ are the same.} \quad ■$$

As a result of the above theorem, we have that strong independence is not only sufficient but also necessary for ensuring preservation of search space. For reordering, we can also extend the results obtained in [4] obtaining:

Definition 6.16 [single solution] A goal g is single solution for constraint c, multi-set of delayed literals D, and program P iff the state $\langle g, c, D \rangle$ has at most one finite, non failure derivation in P. ■

Theorem 6.17 If goal g_2 is both strongly independent of goal g_1 and single solution for constraint c, multi-set of delayed constraints D, and program P in which no atoms can be delayed then $\forall c_1 \in answers_P(\langle g_1, c, D \rangle)$:

$$\#nodes_P(\langle g_2 : g_1, c, D \rangle) \leq \#nodes_P(\langle g_1 : g_2, c, D \rangle). \quad ■$$

6.4 Search Independence

In models designed for shared addressing space machines the isolation of environments is not imposed by the machine architecture and thus, in practice, the goals executing in parallel generally share a single binding environment. Therefore, we need to define a symmetric notion of strong independence:

Definition 6.18 [search independence] Goals g_1 and g_2 are search independent for constraint c, multi-set of delayed constraints D, and program P in which no atoms can be delayed iff

$$\forall \langle c_1, D_1 \rangle \in partial_P(\langle g_1, c, D \rangle) \textit{ and } \forall \langle c_r, D_r \rangle \in partial_P(\langle g_2, c, D \rangle) : consistent(c_s)$$

where c_s is the active constraint obtained by $infer(c_1 \wedge c_r, D_1 \wedge D_r)$. ■

Then we can conclude:

Corollary 6.19 Goals g_1 and g_2 are search independent for constraint c, multi-set of delayed constraints D, and program P in which no atom is allowed to delay iff $\forall \langle c_1, D_1 \rangle \in answers_P(\langle g_1, c, D \rangle)$:

$$\text{the search spaces of } \langle g_2, c, D \rangle \text{ and } \langle g_2, c_1, D_1 \rangle \text{ are the same}$$

and $\forall \langle c_r, D_r \rangle \in answers_P(\langle g_2, c, D \rangle)$:

$$\text{the search spaces of } \langle g_1, c, D \rangle \text{ and } \langle g_1, c_r, D_r \rangle \text{ are the same.} \quad ■$$

7 Second Case: D does not contain atoms

Let us now relax the conditions by allowing atoms to be delayed in the program, but still requiring that the initial multi-set of delayed literals does not contain atoms. We will assume all the conditions imposed on the operational semantics in the previous section, but allowing transitions $\rightarrow_d$ and $\rightarrow_w$. Furthermore, the parametric functions *delay* and *woken* should satisfy the following four conditions. The first ensures that there is a congruence between the conditions for delaying an atom and waking it:

$$(1) \quad a \in woken(D, c) \text{ iff } a \in D \wedge \neg delay(a, c))$$

The remaining conditions ensure that *delay* behaves reasonably. It should not take variable names into account:

$$(2) \quad \text{Let } \rho \in Ren.\ delay(a, c) \text{ iff } delay(\rho(a), \rho(c))$$

It should only be concerned with the effect of c on the variables in a:

$$(3) \quad delay(a, c) \text{ iff } delay(a, \exists_{-vars(a)} c)$$

Also, if an atom is not delayed, adding more constraints should never cause it to delay:

$$(4) \quad \text{If } c \rightarrow c' \text{ and } delay(a, c), \text{ then } delay(a, c')$$

Finally, in most practical systems the order in which atoms are delayed and are woken is important. Thus, for conciseness, we will consider that D is a sequence, rather than a multi-set, and that the order in which literals are added and erased follows the LIFO (last in first out) model.

7.1 Independence and Search Space Preservation

Let us discuss the and-parallel execution model in this new context. Let $atoms(D)$ be the subsequence of atoms in D obtained by eliminating all constraints (note that the relative order among atoms is preserved) and $cons(D)$ be the constraint formed by the conjunction of all primitive constraints in the sequence of literals D. Assume that given the program P and the state $\langle g_1 : g_2 : G, c, D \rangle$ where D is a sequence of constraints, we want to execute g_1 and g_2 in parallel.

Then the execution scheme is the following:

- execute $\langle g_1, c, D \rangle$ and $\langle g_2, c, D \rangle$ in parallel (in different environments) obtaining the answer constraints $\langle c_1, D_1 \rangle$ and $\langle c_r, D_r \rangle$ respectively,
- obtain $(c_s, D_s) = infer(c_1 \wedge c_r, cons(D_1) \wedge cons(D_r))$
- execute $\langle G, c_s, D_r :: D_1 \rangle$.

As a result, the definition of efficiency is the same as that given in the previous section, correctness being extended as follows:

Definition 7.1 Let $\langle g_1 : g_2 : G, c, D \rangle$ be a state in which D is a sequence of constraints and P be a program. The parallel execution of g_1 and g_2 is correct iff for every $\langle c_1, D_1 \rangle \in answers_P(\langle g_1, c, D \rangle)$ there exists a renaming $\rho \in Ren$ and a bijection which assigns to each answer $\langle c_s, D_s \rangle \in answers_P(\langle g_2, c_1, D_1 \rangle)$ an answer $\langle c_r, D_r \rangle \in answers_P(\langle g_2, c, D \rangle)$ with $atoms(D_s) \equiv atoms(\rho(D_r) :: D_1)$ and $(c_s, cons(D_s)) = infer(c_1 \wedge \rho(c_r), cons(D_1) \wedge cons(\rho(D_r)))$. ■

Note that since we require $atoms(D_s) \equiv atoms(\rho(D_r) :: D_1)$, and the state $\langle nil, c_s, D_s \rangle$ is a final state, $\rightarrow_w$ cannot be applied and thus for every atom a in $atoms(\rho(D_r) :: D_1)$, $delay(a, c_s)$ must hold. I.e., every atom left delayed by g_1 must remain delayed in all finite, non failure derivations of $\langle g_2, c, D \rangle$ and every atom left delayed by g_2 must remain delayed in all finite, non failure derivations of $\langle g_2, c_1, D_1\rangle$. Also note that although the constraints woken during the $infer$ operation have not been erased from the sequence of delayed atoms we still have that $(c_s, cons(D_s)) = infer(c_1 \wedge c_r, cons(D_1) \wedge cons(D_r))$

As in the previous section, given the definition of search space preservation, it is straightforward to prove the following result:

Theorem 7.2 Let P be a program, $\langle g_1 : g_2 : G, c, D\rangle$ be a state in which D is a sequence of constraints, and $\langle c_1, D_1\rangle \in answers_P(\langle g_1, c\rangle)$. If the search spaces of $\langle g_2, c, D\rangle$ and $\langle g_2, c_1, D_1\rangle$ are the same for P, then

$$\sum_{i \in TR} K(i) * N(i, \langle g_2, c, D\rangle) = \sum_{i \in TR} K(i) * N(i, \langle g_2, c_1, D_1\rangle). \quad \blacksquare$$

Thus, we can ensure the following:

Theorem 7.3 Let P be a program and $\langle g_1 : g_2 : G, c, D\rangle$ be a state in which D is a sequence of constraints. The parallel execution of g_1 and g_2 is efficient iff for every $\langle c_1, D_1\rangle \in answers_P(\langle g_1, c, D\rangle)$ the search spaces of $\langle g_2, c, D\rangle$ and $\langle g_2, c_1, D_1\rangle$ are the same for P. $\blacksquare$

On the other hand, proving that search space preservation is sufficient for guaranteeing correctness is much more involved. In the previous section, we proved that search space preservation is sufficient for ensuring correctness based on the results of Lemma 6.4 and Lemma 6.5. However, these lemmas do not apply in the new context. Lemma 6.4 guarantees that, in absence of failure, there exists a bijection which assigns to each node r in $tree_P(\langle g_2, c, D\rangle)$ a node s in $tree_P(\langle g_2, c_1, D_1\rangle)$ with the same path, the sequence of active literals in such nodes being identical up to renaming. This result was used in Lemma 6.5 to ensure that, in absence of failure, the constraints selected by the computation rule processed in $tree_P(\langle g_2, c, D\rangle)$ and in $tree_P(\langle g_2, c_1, D_1\rangle)$ are the same, up to renaming. Unfortunately, in the new context we cannot guarantee that the same atoms are woken in both executions, and thus the sequence of active literals in nodes with the same path can differ.

Example 7.1 Consider the state $\langle \texttt{p(y)} : \texttt{q(x,y)}, \epsilon, nil\rangle$ and the program P:

```
p(x,y)     <-  y = x.
q(x,y)     <-  r(x), s(x,y,w), t(w), y = 0.
r(x)       <-  x = 0.
s(x,y,w)   <-  w = f(x,z), w = f(0,1).
```

with the following suspension declarations for `p/2, q/2, r/1` and `s/2`:

```
?- r(x) when ground(x).
?- s(x,y) when ground(y).
```

Figure 2 shows (a) the derivation tree from the state $\langle \texttt{q(x,y)}, \epsilon, nil\rangle$ (i.e., `p(x,y)` has not been executed) and (b) the derivation tree from the state $\langle \texttt{q(x,y)}, \texttt{y = x}, \texttt{nil}\rangle$ (i.e., `p(x,y)` has been executed obtaining in the answer $\langle \texttt{x = y}, nil\rangle$). It is clear that even though search space is preserved, there is no renaming which makes the sequences of active literals identical. Furthermore, there is also no renaming which makes identical the leftmost literal of every two non failure nodes with the same path. Finally, there is a leftmost literal `r(x)` which is not selected to be processed.□

The main problem is that, even while search space is still being preserved, relatively different executions can happen. Firstly, an atom a in D_1 can be woken during the execution of $\langle g_2, c_1, D_1\rangle$ at state $s_a\langle a : G, c_a, D_a\rangle$ without changing the search space, if all branches in the tree of s_a are failure, and the tree starting from the node in $tree_P(g_2, c, D)$ with the same path of s_a, has the same search space. Note that

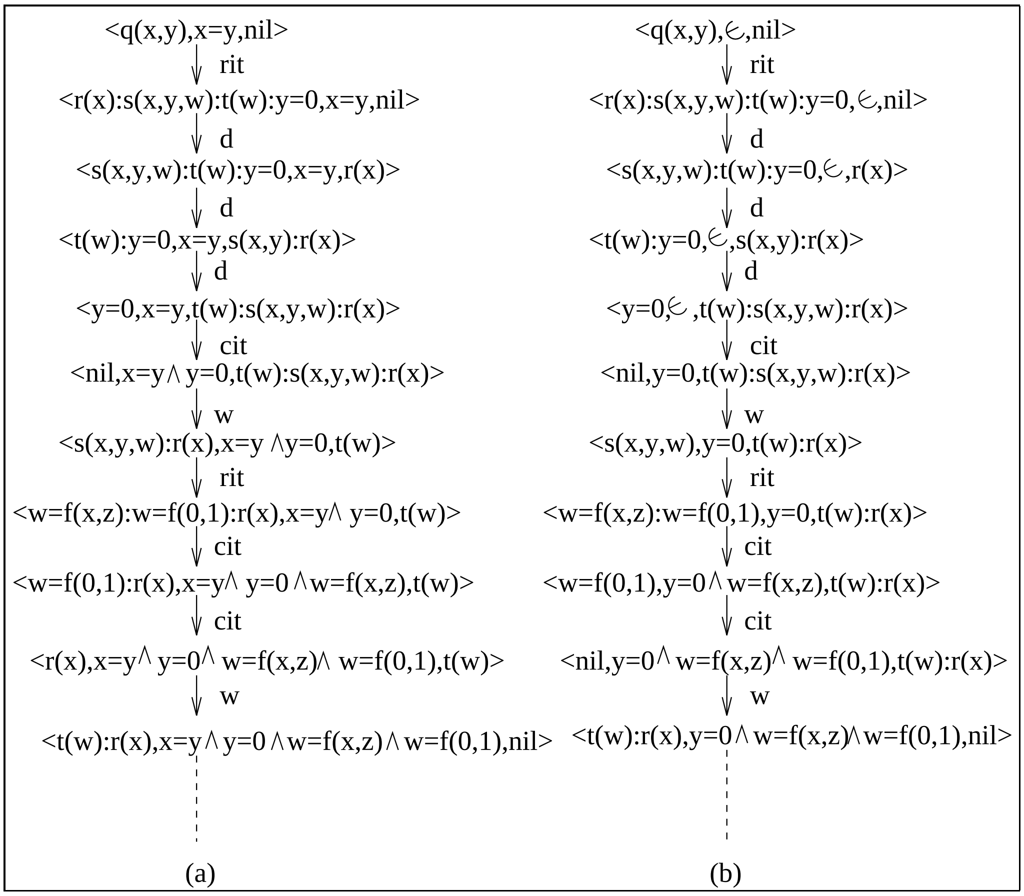

Figure 2:

correctness would not be affected since we are talking about failure branches. Secondly, during the execution of $\langle g_2, c, D\rangle$ and $\langle g_2, c_1, D_1\rangle$, the same atoms can be woken in different order, but still preserving the search space and the correctness. Therefore, the same literal can be processed once in $\langle g_2, c, D\rangle$ and several times in $\langle g_2, c_1, D_1\rangle$ or vice versa, and obviously not in nodes with the same path. Note that this can be related to the notion of interleavings in concurrent languages. In some sense, this result is not surprising since dynamically scheduling is similar to concurrency, and therefore interleavings have to be taken into account.

Even with all these problems, we can still prove that search space preservation is *sufficient for preserving correctness*. The intuition behind this fact is that for finite, non failure branches, every literal processed in $tree_P(\langle g_2, c, D\rangle)$ will be *sooner or later* processed in $tree_P(\langle g_1, c_1, D_1\rangle)$ and vice versa. Thus, if search space is preserved, every atom left delayed by g_1 must remain delayed in all finite, non failure derivations of $\langle g_2, c, D\rangle$ and every atom left delayed by g_2 must remain delayed in all finite, non failure derivations of $\langle g_2, c_1, D_1\rangle$. On the other hand, it is not longer necessary for ensuring that both correctness and efficiency hold for the parallel execution of the goals, since the failure branches can be both enlarged and pruned involving the same cost without preserving the search space. As a result, it is not longer true either that search space preservation is necessary for ensuring that the number of nodes in the trees are the same. Thus, search space preservation and preservation of the number of nodes cannot be identified any more.

However, proving that search space preservation is sufficient for guaranteeing correctness is long and tedious. We will avoid such exercise due to the existence of another problem which, when solved, will simplify the results. This problem is that although search space preservation guarantees the existence of a bijection between answers, it cannot guarantee that the order in which the sequential answers are obtained will be preserved when the goals are executed in parallel. This is a desirable property when parallelizing a program, since it guarantees that the order intended by the programmer is preserved. In the context of Section 6, this preservation comes for free due to the existence of a bijection between answers associated to nodes with the same path. In this new context, the bijection does not necessary apply between such nodes, due to the possible existence of interleavings involving goals which are not single solution.

We can avoid such interleavings by ensuring that for every answer $\langle c_1, D_1\rangle$ of $\langle g_1, c, D\rangle$, no atom in D_1 is woken during the execution of $\langle g_2, c_1, D_1\rangle$, and every atom left delayed (woken) at some point of the execution of $\langle g_2, c_1, D_1\rangle$ is also left delayed (woken) at the same point of the execution of $\langle g_2, c, D\rangle$. Formally, we will say that a node $s \equiv \langle G_s, c_s, D_s\rangle$ is *equivalent w.r.t. delay* to a node $r \equiv \langle G_r, c_r, D_r\rangle$ if for every $a \in atoms(D_r)$: $delay(a, c_r)$ iff $delay(a, c_s)$, and for every $a \in atoms(D_s \setminus D_r)$: $delay(a, c_s)$ holds. We will require this condition to be satisfied for every two nodes s and r of $tree_P(\langle g_2, c_1, D_1\rangle)$ and $\rho(tree_P(\langle g_2, c, D\rangle))$, respectively, with the same path. Note that this condition is not necessary since it does not allow the interleaving even when one of the goals involved is single solution or it affects branches other than non failure, finite branches. However, if such condition is satisfied, the situation becomes equivalent to the previous one, allowing us to extend all results obtained in the previous section to this new context.

Lemma 7.4 Let P be a program and $\langle g_2, c_1, D_1\rangle$, $\langle g_2, c, D\rangle$ be two states with an identical sequence of active literals in which D is a sequence of constraints. There exists a renaming $\rho \in Ren$ such that for every two non failure nodes $s \equiv \langle G_s, c_s, D_s\rangle$ and $r \equiv \langle G_r, c_r, D_r\rangle$ with the same path in $tree_P(\langle g_2, c_1, D\rangle)$ and $\rho(tree_P(\langle g_2, c, D\rangle)$, respectively, such that for all ascendants s' and r' of s and r, respectively, with the same path s' is equivalent w.r.t. delay to r' : $G_s \equiv G_r$ and $atoms(D_r : D_1) \equiv atoms(D_s)$. ■

Lemma 7.5 Let $\langle g_2, c_1, D_1\rangle$ and $\langle g_2, c, D\rangle$ be two states with an identical sequence of literals such that D is a sequence of constraints, $c_1 \to c$ and $c_1 \wedge cons(D_1) \to cons(D)$. Let P be a program and $\rho \in Ren$ be a renaming satisfying Lemma 7.4. Then, for every two nodes $s \equiv \langle G_s, c_s, D_s\rangle$ and $r \equiv \langle G_r, c_r, D_r\rangle$ with the same path in $tree_P(\langle g_2, c_1, D_1\rangle)$ and $\rho(tree_P(\langle g_2, c, D\rangle)$, respectively, such that for all their ascendants s' and r', respectively, with the same path s' is equivalent w.r.t. delay to r': $(c_s, D_s) = infer(c_1 \wedge c_r, D_1 \wedge D_r)$. ■

Given the above results, we can ensure the following:

Theorem 7.6 Let P be a program, $\langle g_1 : g_2 : G, c, D\rangle$ be a state in which D is a sequence of constraints, and ρ be a renaming satisfying Lemma 7.4. If for every $\langle c_1, D_1\rangle \in answers_P(\langle g_1, c, D\rangle)$, the search spaces of $\langle g_2, c, D\rangle$ and $\langle g_2, c_1, D_1\rangle$ are the same for P and for every two non failure nodes s and r with the same path in $tree_P(\langle g_2, c_1, D\rangle)$ and $\rho(tree_P(\langle g_2, c, D\rangle))$, s is equivalent w.r.t. delay to r, then there exists a bijection which assigns to each final state $\langle nil, c_s, D_s\rangle$ in $tree_P(\langle g_2, c_1, D_1\rangle)$ a final state $\langle nil, c_r, D_r\rangle$ in $tree_P(\langle g_2, c, D\rangle)$ with the same path such that $atoms(D_s) \equiv atoms(\rho(D_r) :: D_1)$ and $(c_s, D_s) = infer(c_1 \wedge \rho(c_r), cons(D_1) \wedge cons(\rho(D_r)))$. ■

7.2 Weak Independence

As mentioned before, weak independence aims at characterizing those goals for which an independence-based form of intelligent backtracking can be safely performed. Interestingly, in this new context this characterization does not even need that the preservation of the search space among finite, non failure branches hold. We just need that the conjunction of the answers be consistent. The motivation behind this fact is that, no matter if goals are delayed or woken in different order, if the answers are consistent, it is straightforward to prove that if $answers_P(\langle g_i, c_1, D_1\rangle) = \emptyset$ and the sequential execution is able to detect it (i.e., it does not enter in an infinite branch) then $answers_P(\langle g_i, c, D\rangle) = \emptyset$. Thus, for every $\langle c_2, D_2\rangle \in answers_P(\langle g_1 : \cdots : g_{i-1}, c, D\rangle)$: $answers_P(\langle g_i, c_2, D_2\rangle) = \emptyset$.

As a result, the definition of weakly independent goals is identical to Definition 6.11, but allowing the program P to define literals which can be woken and delayed. Analogously, the results obtained for this kind of goals can be extended to this new context in a straightforward way.

7.3 Strong Independence

In this context, strong independence is aimed at detecting goals whose parallelization, when executed in different environments, is guaranteed to be correct, efficient, *and preserves the order among answers.* Thus the definition we are looking for is the following:

Definition 7.7 [strong independence] Goal g_2 is strongly independent of goal g_1 for constraint c, sequence of delayed constraints D, and program P iff:

$$\forall \langle c_1, D_1 \rangle \in answer_P(\langle g_1, c, D \rangle). \forall \langle c_r, D_r \rangle \in partial_P(\langle g_2, c, D \rangle):$$

1. $consistent(c_s)$ holds
2. $\langle nil, c_s, D_1 \rangle$ is equivalent w.r.t delay to $\langle nil, c_r, D_r \rangle$

where c_s is the active constraint returned by $infer(c_1 \wedge c_r, cons(D_1 :: D_r))$. ■

The definition can be extended to a set of goals analogously to Definition 6.14. Thanks to condition (2) and Theorem 6.15, it is straightforward to prove not only that strong independence implies search space preservation, but also that the bijection required for correctness holds for nodes with the same path:

Theorem 7.8 If Goal g_2 is strongly independent of goal g_1 for constraint c, multi-set of delayed constraints D, and program P, then $\forall \langle c_1, D_1 \rangle \in answers_P(\langle g_1, c, D \rangle)$:

- the search spaces of $\langle g_2, c, D \rangle$ and $\langle g_2, c_1, D_1 \rangle$ are the same, and
- there exists a renaming $\rho \in Ren$ and a bijection which assigns to each final state $\langle nil, c_s, D_s \rangle$ in $tree_P(\langle g_2, c_1, D_1 \rangle)$ a final state $\langle nil, c_r, D_r \rangle$ in $tree_P(\langle g_2, c, D \rangle)$ with the same path such that $atoms(D_s) \equiv atoms(\rho(D_r) :: D_1)$ and $(c_s, D_s) = infer(c_1 \wedge \rho(c_r), cons(D_1) \wedge cons(\rho(D_r)))$. ■

On the other hand, Theorem 6.17 does not apply in this new context since, even if goal g_2 is both strongly independent of goal g_1 and single solution for constraint c, multi-set of delayed literals D in which $atoms(D) \equiv nil$, and program P, there may exist $c_1 \in answers_P(\langle g_1, c, D \rangle)$ for which $\#nodes_P(\langle g_2 : g_1, c, D \rangle) > \#nodes_P(\langle g_1 : g_2, c, D \rangle)$. For example, if there exists $\langle c_r, D_r \rangle \in answers_P(\langle g_2, c, D \rangle)$, such that an atom in D_r is woken in a failure derivation of $\langle g_1, c_r, D_r \rangle$, and it is not woken during the execution of $\langle g_1, c, D \rangle$, the number of non failure nodes can be greater in $\langle g_2 : g_1, c, D \rangle$. As a result, we will need the symmetric concept of independence that will be developed in the next section. Given that the number of non failure nodes will be in this case identical after reordering, speed up can only be obtained from this transformation if the amount of work when adding the constraints to the store is in any way reduced.

7.4 Search Independence

As mentioned before, in models designed for shared addressing space machines the isolation of environments is not imposed by the machine architecture and thus, in practice, the goals executing in parallel generally share a single binding environment. However, in the context of dynamically scheduled languages, it is useful to require that the stack modeling the sequence of delayed atoms remain isolated for each parallel goal. The motivation for this is to allow each parallel agent to easily recognize the atoms left delayed during each parallel execution, and also to simplify the execution in distributed environments.

In this context, we cannot allow an atom in the local stack of a parallel goal to be woken by the constraints added by other parallel execution. Thus, we should extend the definition of search independence as follows:

Definition 7.9 [search independence] Goals g_1 and g_2 are search independent for constraint c, multi-set of delayed constraints D, and program P in which no atoms can be delayed iff

$$\forall \langle c_1, D_1 \rangle \in partial_P(\langle g_1, c, D \rangle). \forall \langle c_r, D_r \rangle \in partial_P(\langle g_2, c, D \rangle):$$

1. $consistent(c_s)$ holds

2. for every $a \in atoms(D_r)$: $delay(a, c_r)$ iff $delay(a, c_s)$

3. for every $a \in atoms(D_1)$: $delay(a, c_1)$ iff $delay(a, c_s)$

where c_s is the active constraint returned by $infer(c_1 \wedge c_r, cons(D_1 :: D_r))$. ■

Note that the last two conditions imply a symmetric notion of equivalence w.r.t. delay. All results obtained in the previous section for search independent goals, can be extended to this new context in a straightforward way.

8 General Case

Let us now consider the more general case in which the sequence of delayed literals D might contain atoms. Two problems appear in this new context. The first problem is related to the definition of the and-parallel model and, in particular, to the "conjoin" operation. This operation – conjoining the sequence of delayed atoms associated to the answers obtained in the parallel execution – must be done in such a way that the resulting sequence preserves the order among atoms established by the sequential execution. The existence of atoms in the initial sequence of delayed literals increases the complexity of such operation. Note that the order among delayed constraints is not important. Furthermore, as mentioned before, it is not even necessary to eliminate the woken constraints from the conjoined sequence of delayed literals, although it is convenient for efficiency reasons. The second problem is related to the existence of atoms in D which can be woken by both g_1 and g_2. This problem can easily be solved by extending the definition of equivalence w.r.t. delay to the following: node $s \equiv \langle G_s, c_s, D_s \rangle$ is *equivalent w.r.t. delay* to node $r \equiv \langle G_r, c_r, D_r \rangle$ if, for every $a \in atoms(D_r)$: $delay(a, c_r)$ iff $delay(a, c_s)$, for every $a \in atoms(D_s \setminus D_r)$: $delay(a, c_s)$ holds, and for every $a \in atoms(D_r \setminus D_s)$: $delay(a, c_r)$ holds. I.e., the idea is to require also that all atoms in D not present in D_1 remain delayed during the execution of $\langle g_2, c, D \rangle$.

This solution allows us to solve the first problem by defining the conjoin operation as follows: D_s is obtained in the and-parallel model as

$$(D_r \setminus atoms(D)) :: (D_1 \setminus (atoms(D \setminus D_r)))$$

The intuition behind the above operation is that we have to eliminate from D_1 the atoms woken by $\langle g_2, c, D \rangle$ (represented by $atoms(D \setminus D_r)$) and then add the atoms left delayed by $\langle g_2, c, D \rangle$ which do not belong to the initial sequence (represented by $(D_r \setminus atoms(D))$).

With this solution the results obtained in the previous sections regarding the characteristics of both the search space preservation and the levels of independence can be extended to this new context in a straightforward way. Note that the definition of search space given in the previous section remains the same. This is because the two last conditions of Definition 7.9 already imply the extended notion of equivalence w.r.t. delay proposed above.

9 Ensuring Search Independence "A Priori"

It is important to determine sufficient conditions which ensure search independence from just the information given by the store rather than from the information given by the partial answers. There are two main reasons for this. First, such sufficient conditions could be tested at run-time just before executing the goals without actually having to execute them (we refer to this as "a priori" detection of independence). Second, the kind of global data-flow analysis required for inferring the information needed to ensure that these sufficient conditions hold may be less complex than that needed for ensuring directly the definition of search independence.

In [4] we have shown that, in the context of CLP languages, goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are search independent for a given constraint c if they are *projection independent*, i.e., iff

$$(\bar{x} \cap \bar{y} \subseteq def(c)) \ and \ (\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c \rightarrow \ \exists_{-\bar{y} \cup \bar{x}} c)$$

where $def(c)$ denotes the set of variables constrained to a unique value in c. As before, note that $(\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c \leftarrow \ \exists_{-\bar{y} \cup \bar{x}} c)$ is always satisfied.

Intuitively, the condition states that (a) the goals do not have variables in common w.r.t. c and (b) by projecting the constraint store c over the variables of each goal we do not lose "interesting" information w.r.t. projecting the original constraint store over the variables of both goals (i.e., the store obtained in the former way entails that obtained in the latter way). This ensures that no matter how $g_1(\bar{x})$ and $g_2(\bar{y})$ are defined, the execution of one goal will not be able to modify the domain of the variables of the other, and vice-versa.

Let us now discuss this sufficient condition in the broader context of languages with dynamic scheduling. Consider two goals g_1 and g_2 and a given constraint c for which the above condition is satisfied. If the sequence of delayed literals D is empty (note that now neither atoms nor constraints are allowed), then we can ensure that the goals are search independent by simply detecting that the above condition holds. The intuition behind this fact is that if there are no delayed literals before the execution of the goals, and they cannot affect the domain of each other's variables, then their partial answers will be consistent and the instantiation state of their variables will not change no matter if one is executed before or after the other, thus not affecting the literals left delayed by the other goal. Formally:

Theorem 9.1 Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are search independent for a given constraint c and a sequence of delayed atoms $D \equiv nil$, if

$$(\bar{x} \cap \bar{y} \subseteq def(c)) \ and \ (\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c \rightarrow \ \exists_{-\bar{y} \cup \bar{x}} c) \quad \blacksquare$$

A difference w.r.t. previous cases does arise, however, when the sequence of delayed literals just before the execution of the goals is not empty. In this case the sufficient condition must take into account the constraints established on the variables which appear in the sequence of delayed literals. The reason is that literals woken during the execution of either $g_1(\bar{x})$ or $g_2(\bar{y})$ may introduce new constraints involving variables in both $\bar{x}$ and $\bar{y}$.

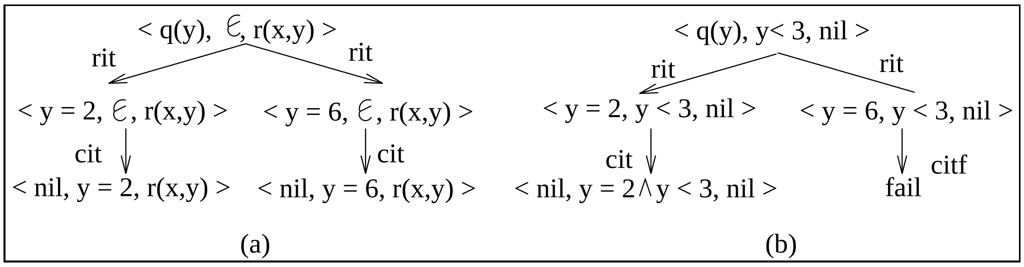

Figure 3:

Example 9.1 Consider the following simple example:

```
p(x)    ←  x = 3.

q(y)    ←  y = 6.
q(y)    ←  y = 2.
```

$\mathtt{r(x,y)} \quad \leftarrow \quad \mathtt{x < y}.$

in which `r(x,y)` will be delayed until `x` becomes constrained to a unique value.

Figure 3 shows (a) the derivations from the state $\langle \mathtt{q(y)}, \epsilon, \mathtt{r(x,y)} \rangle$ (i.e., `p(x)` has not been executed) and (b) the derivations from the state $\langle \mathtt{q(y)}, \mathtt{x} = 3 \wedge \mathtt{y} < \mathtt{x}, nil \rangle$ (i.e., `p(x)` has been executed waking `r(x,y)`). It is clear that although `p(x)` and `q(y)` satisfy the sufficient condition w.r.t. ϵ and the empty sequence of delayed atoms, if `r(x,y)` is woken before the execution of `q(y)` it will prune the search space of `q(y)` by making the branch corresponding to `y = 6` fail. □

A similar situation may appear when constraints become active. Thus, new conditions must be developed in order to take the sequence of delayed atoms into account. The solution proposed is to ensure that D can be partitioned into two sequences in such a way that if we associate those sequences to $g_1(\bar{x})$ and $g_2(\bar{y})$ respectively, the two new goals satisfy the sufficient condition for the given c and an empty sequence of delayed literals. While the first sequence corresponds to the delayed goals that depend on $g_1(\bar{x})$, the second one corresponds to those that depend on $g_2(\bar{y})$. If there exist delayed literals which depend on neither $g_1(\bar{x})$ nor $g_2(\bar{y})$, they can be concatenated to any of them.

Definition 9.2 [projection independence] Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for constraint c and sequence of delayed literals D iff D can be partitioned into two sequences D_1 and D_2 such that the goal $g_1(\bar{x}) : D_1$ and $g_2(\bar{y}) : D_2$ are projection independent for c and the empty sequence of delayed literals.

Theorem 9.3 Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are search independent for constraint c and sequence of delayed goals D if they are projection independent for c and D. ■

Note that again, when considering CLP languages without dynamic scheduling, this definition is identical to that defined in [4]. Furthermore, we argue that all sufficient conditions given in [4] are directly applicable to languages with dynamic scheduling by simply transforming the given sequences of delayed literals as proposed above.

10 Solver Independence

In the context of CLP languages, search space preservation is not enough for ensuring the efficiency of any transformation applied to the search independent goals. The reason is that modifying the order in which a sequence of primitive constraints is added to the store may have a critical influence on the time spent by the constraint solver algorithm in obtaining the answer, even if the resulting constraint is consistent. For this reason a new type of independence, *constraint solver independence*, was defined.

This concept is orthogonal to the issue of search space preservation and is only related to the characteristics of the particular constraint solver considered when adding sequences of primitive constraints in different orders. Thus it is tempting to think that the definitions and results obtained for CLP languages in [4] can be directly applied to languages with dynamic scheduling. However, there is one question which must be considered: the work involved in determining if a goal must become delayed or must be woken.

We believe that a similar approach to that of [4] can be taken, by considering the solvers which are independent in this sense, those which require the parallel goals, g_1 and g_2, to be projection independent for the store c and the sequence of delayed literals D, and those which also require the goals to be link independent for c and D. However, this issue certainly needs further study.

References

[1] K.R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Modeel and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] 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.

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

[4] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.

[5] S. K. Debray. QD-Janus : A Sequential Implementation of Janus in Prolog. *Software—Practice and Experience*, 23(12):1337–1360, December 1993.

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

[7] 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.

[8] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[9] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 1994. To appear.

[10] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[11] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *ACM Symp. Principles of Programming Languages*, pages 111–119. ACM, 1987.

[12] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming, to appear*, 1994.

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

[14] J. W. Lloyd. *Logic Programming.* Springer-Verlag, 1987.

[15] V. Saraswat. Compiling CP() on top of Prolog. Technical Report CMU-CS-87-174, Computer Science Department, Carnegie-Mellon University, Pittsburgh, October 1987.

[16] K. Ueda and T. Chikiyama. A compiler for concurrent prolog. In *Second International Symposium on Logic Programming*, pages 119–126, Boston, July (1985). IEEE Press.

[17] 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.

[18] 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.