Independence in CLP Languages

MARÍA GARCÍA DE LA BANDA
Monash University
MANUEL HERMENEGILDO
Technical University of Madrid (UPM)
and
KIM MARRIOTT
Monash University

Studying independence of goals has proven very useful in the context of logic programming. In particular, it has provided a formal basis for powerful automatic parallelization tools, since independence ensures that two goals may be evaluated in parallel while preserving correctness and efficiency. We extend the concept of independence to constraint logic programs (CLP) and prove that it also ensures the correctness and efficiency of the parallel evaluation of independent goals. Independence for CLP languages is more complex than for logic programming as search space preservation is necessary but no longer sufficient for ensuring correctness and efficiency. Two additional issues arise. The first is that the cost of constraint solving may depend upon the order constraints are encountered. The second is the need to handle dynamic scheduling. We clarify these issues by proposing various types of search independence and constraint solver independence, and show how they can be combined to allow different optimizations, from parallelism to intelligent backtracking. Sufficient conditions for independence which can be evaluated "a priori" at run-time are also proposed. Our study also yields new insights into independence in logic programming languages. In particular, we show that search space preservation is not only a sufficient but also a necessary condition for ensuring correctness and efficiency of parallel execution.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming—*automatic analysis of algorithms; program transformation*; D.1.3 [**Programming Techniques**]: Parallel Programming; D.1.6 [**Programming Techniques**]: Logic Programming; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs*

General Terms: Languages, Performance

Additional Key Words and Phrases: Constraint logic programming, independence, parallelism

This work was funded in part by ESPRIT projects 7195 "ACCLAIM" and 5246 "PRINCE", and by CICYT projects TIC93-0975-CE, TIC91-0106-CE, and TIC99-1151 "*EDIPIA*". M. García de la Banda was supported by a grant from the Australian Research Council and by a Logan Fellowship from Monash University. Preliminary versions of different parts of this paper were presented at the 1993 International Logic Programming Symposium, pages 130–146, MIT Press, and at the 1996 International Conference on Algebraic and Logic Programming, pages 47–61, Springer-Verlag.
Authors' addresses: M. Hermenegildo, Universidad Politécnica de Madrid, Facultad de Informática, Madrid, Spain; email: herme@fi.upm.es; M. García de la Banda and K. Marriott, Monash University, Computer Science, Melbourne, Australia; email: {mbanda,marriott}@cs.monash.edu.au.

1 INTRODUCTION

The notion of *independence* of program statements or procedure calls is relatively well understood in the context of imperative languages, where several definitions of independence, ranging from those based on the Bernstein conditions to more recent notions of "semantic independence," have been defined and applied primarily in program parallelization [Bacon et al. 1994; Best and Lengauer 1990]. Independence has also been studied and proved to be a very useful concept in traditional logic programming. Again, the primary motivation is program parallelization [Hermenegildo and Rossi 1995; Haridi and Janson 1990]. However, it also provides a theoretical basis for other powerful program optimizations, including intelligent backtracking [Pereira and Porto 1982], and goal reordering [Warren and Pereira 1982].

The general, intuitive notion of independence in logic programming is that a goal q is independent of a goal p if p does not "affect" q. A goal p is understood to affect another goal q if p changes the execution of q in an "observable" way. Observables include changing the solutions that q produces ("correctness") and changing the time that it takes to compute such solutions ("efficiency"). This contrasts with more traditional notions of independence which, because of the characteristics of imperative or functional languages, only need to deal with the preservation of correctness [Hermenegildo 1997].

Previous work in the context of traditional logic programming languages [Conery 1983; DeGroot 1984; Hermenegildo and Rossi 1995; Chassin and Codognet 1994] has concentrated on defining sufficient conditions which ensure that goals can be safely executed in parallel. This has been achieved by ensuring that either the goals do not share variables (*strict independence*) or if they share variables, that they do not "compete" for their bindings (*non strict independence*).

In this paper we consider independence in the general context of the constraint logic programming (CLP) paradigm [Jaffar and Lassez 1987], which has emerged as the natural combination of the constraint solving and logic programming paradigms. As for logic programming, our main motivation is to find conditions which allow goals to be executed in parallel. However, we shall also investigate other types of independence, each of which is "interesting" for a certain class of program transformations.

Generalizing the independence results obtained for logic programming to CLP is difficult for two reasons. The first reason is that the cost of constraint solving may depend upon the order in which constraints are encountered. This means we need to introduce a notion of "constraint solver independence" which captures how sensitive the solver is to reordering of constraints. This issue did not arise for logic programs because the standard unification algorithm, as usually implemented, is, in most practical cases, independent in this sense. However, in the more general context of CLP, constraint solver independence need not hold. The second reason is that many CLP languages provide dynamic scheduling of literals in goals. This is useful because it facilitates definition or extension of constraint solvers but is considerably more difficult to understand than the standard left-to-right evaluation of goals in logic programs. Actually, dynamic scheduling is also present in some logic programming languages, but since it is not widely used it has been ignored in work on parallelization. However, it must be addressed in the CLP context because of its importance when writing constraint solvers.

Generalizing independence to arbitrary CLP languages and constraint solvers is not only interesting in itself, but also yields new insights into independence even for logic programs. First, it allows us to simplify many of the earlier results by couching them in terms of constraints rather than substitutions. Second, extension of the results to the case of dynamic scheduling has required us to precisely formalize search space preservation and its relationship to independence.

We believe that generalization of independence to CLP will be useful, since the associated optimizations performed in the context of logic programming appear equally applicable to the context of constraints. Indeed, the cost of performing constraint satisfaction makes the potential performance improvements even larger. Preliminary experiments with and-parallelization of CLP [García de la Banda et al. 1996] provide some evidence in this direction.

The rest of the paper proceeds as follows. Section 2 reviews various models for the parallel execution of logic programs and the associated notions of independence. Section 3 formally defines a parallel execution model for CLP programs. Section 4 clarifies the relationship between search space preservation and the safety of parallel execution. Section 5 presents several concepts of independence for CLP, each one useful for a class of applications and relates these to search space preservation. Section 6 gives sufficient conditions that are easier to detect at run-time than the definitions of independence. Section 7 discusses the notion of independence for CLP at the solver level and discusses additional characteristics required of the solvers, offering some examples. Section 8 extends these results to CLP languages that provide dynamic scheduling. Finally, Section 9 presents our conclusions.

2 INDEPENDENCE FOR PARALLELIZATION IN LOGIC PROGRAMS REVISITED

2.1 Operational Semantics of Logic Programs

In this section we introduce some basic concepts and notation regarding logic programs. We will follow mainly [Apt 1990; Lloyd 1987]. Note that we will only deal with *definite* logic programs (also referred to as *positive* logic programs). Also note that while the math italics font will be used for definitions and theorems to represent general objects, the teletype font will be used for representing particular instances of the objects, such as those coming from an example program.

An *atom* has the form $p(\bar{x})$ where $\bar{x}$ is a sequence of distinct variables and p is a predicate symbol. An *equation* has the form $t = u$ where t and u are terms. A *literal* is an atom or an equation. A *clause* or *rule* has the form $h \leftarrow b_1, \cdots, b_n$ with $n \geq 0$, where h is an atom called the *head* and $b_1, \cdots, b_n$ is a sequence of literals called the *body*. A *program* is a set of rules. A *goal* is a sequence of literals. The empty literal sequence is denoted by nil, and often omitted. We let $vars(t)$ denote the set of variables occurring in a syntactic expression t. A syntactic expression t is *ground* if $vars(t) = \emptyset$. The *local variables* of the clause $h \leftarrow b_1, \cdots, b_n$ are those variables appearing in the body but not in the head, i.e., $(vars(b_1) \cup \cdots \cup vars(b_n)) \setminus vars(h)$.

A *renaming* is a bijective mapping from variables to variables. We naturally extend renamings to mappings between syntactic objects. Syntactic objects s and s' are said to be *variants* if there is a renaming such that $\rho(s) \equiv s'$ where $\equiv$ denotes syntactic equivalence.

The operational semantics of logic programs is couched in terms of substitutions. A *substitution* is a (finite) mapping from variables to terms, and it is represented as $\{x_1/t_1, \cdots, x_n/t_n\}$. The domain of a substitution $\theta = \{x_1/t_1, \cdots, x_n/t_n\}$ is denoted by $dom(\theta)$ and defined as $\{x_1, \cdots, x_n\}$. Its range, is denoted by $range(\theta)$ and defined as $vars(t_1) \cup \cdots \cup vars(t_n)$. A pair x/t is called a *binding*. We assume that for each binding x/t in a substitution, $x \not\equiv t$. The empty substitution is denoted ϵ. The application of a substitution θ to a syntactic object s is denoted by $s\theta$ and it is defined to be the syntactic object obtained by replacing each variable x in s by $\theta(x)$. Composition of substitutions θ and σ is defined as function composition and denoted $\theta\sigma$, so that for any syntactic object s we have $s\theta\sigma = (s\theta)\sigma$, i.e., θ is applied first. A substitution θ' is *more general* than θ, written $\theta \leq \theta'$, iff there exists another substitution σ such that $\theta = \theta'\sigma$. A substitution θ is *idempotent* if $\theta\theta = \theta$. We shall only be interested in idempotent substitutions.

A variable x is *ground* with respect to a substitution θ if $\theta(x)$ is ground. A set of variables $\{x_1, \cdots, x_n\}$ are *aliased* or *share* with respect to a substitution θ if $vars(\theta(x_1)) \cap \cdots \cap vars(\theta(x_n)) \neq \emptyset$.

Substitutions are used to represent the solutions to term equations. A substitution θ is a *unifier* of an equation $e \equiv t = u$ iff $t\theta \equiv u\theta$. If such a unifier exists, e is said to be *unifiable*. A substitution θ is a *most general unifier* of e iff θ is more general than any other unifier of e. If e has a most general unifier, it has an idempotent most general unifier. A set of equations $\{x_1 = t_1, \cdots, x_n = t_n\}$ is in *solved form* if each x_i is a distinct variable and $\{x_1, \cdots, x_n\}$ is disjoint from $vars(t_1) \cup \cdots \cup vars(t_n)$. The solved form of an equation e is given by a set $Solv \equiv \{x_1 = t_1, \cdots, x_n = t_n\}$, such that $Solv$ is in solved form, $vars(e) \subseteq \{x_1, \cdots, x_n\}$ and e is equivalent to the conjunction of the equations in $Solv$. Note that all most general unifiers of an equation are equivalent and essentially represent the solved form of the equation. The function mgu returns an idempotent most general unifier of a term equation if it exists. Otherwise it fails.

Logic programs are evaluated through a combination of two mechanisms: replacement and unification. This strategy is named *SLD-resolution*. The operational semantics of a program P can be presented as a transition on *states* $\langle G, \theta \rangle$, where G is a goal, and θ is a substitution. The semantics is parameterized by a *computation rule* and a *search rule*. A computation rule selects a transition rule and an appropriate element of G in each state. A search rule selects a given clause of the program. For simplicity, we use the standard left-to-right computation rule and depth first search strategy (as used in Prolog).

Let a be an atom and e an equation. The transition rules are as follows. Note that the conditions for applying each of the transition rules are pairwise exclusive.

- $\langle a : G, \theta \rangle \rightarrow \langle B : G, \theta \rangle$ if $B \in defn_P(a)$;
- $\langle a : G, \theta \rangle \rightarrow fail$ if $defn_P(a) = \emptyset$;
- $\langle e : G, \theta \rangle \rightarrow \langle G, \theta\theta' \rangle$ if $mgu(e\theta) = \theta'$;
- $\langle e : G, \theta \rangle \rightarrow fail$ if $mgu(e\theta)$ fails.

We let $defn_P(a)$ denote the *definition of atom a in program P*. This is the set of appropriately renamed rule bodies in P whose corresponding rule head is a variant of a. More exactly,

$$defn_P(a) = \{\rho_{a,h}(B) | h \leftarrow B \in P\}$$

where each renaming $\rho_{a,h}$ is chosen so that $\rho(h) \equiv a$ and where the local variables in B are renamed to new variables never seen before in any other transition step.

A *derivation* of a state s 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 is a *final state*. A derivation is *successful* when it is finite and the final state has the form $\langle nil, \theta\rangle$. A derivation is *failed* when it is finite and the final state is $fail$. The substitution θ is said to be a *partial answer* to state s if there is a derivation from s to a state $\langle G, \theta\rangle$ and it is said to be an *answer* if $\langle G, \theta\rangle$ is a final state (i.e., $G \equiv nil$).

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 for state s and program P, denoted by $tree_P(s)$, represents the search space for finding all answers to s and is unique up to 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*.

2.2 Independence for Parallelization in Logic Programs

This section provides a brief history of the various notions of independence developed in the context of traditional logic programming. Consequently none of the definitions of independence in this section are new; rather this review of earlier work provides the necessary background for our research and allows us to clarify our contribution.

The several independence notions defined in the context of traditional logic programming were generally developed for the particular application of program parallelization within the independent and-parallelism model [Conery 1983; DeGroot 1984; Hermenegildo and Rossi 1995]. This model aims at running in parallel as many "independent" goals as possible while maintaining correctness and efficiency with respect to the sequential execution where independence between goals implies that they have no communication between them and that they may be run in different environments.

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 than or equal to the sequential execution time. This was approximated by requiring that the amount of work performed for computing the answers during the parallel execution be no more than that performed in the sequential execution.

In this context, 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.

Assume that we are given the state $\langle g_1 : g_2 : G, \theta\rangle$ and wish to execute g_1 and g_2 in parallel (the extension to sequences of *consecutive* goals is straightforward). One possible execution model is to

- 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, and
- execute $\langle G, \theta_1\theta_2\rangle$.

This model was intended to be generic, abstracting away from implementation details such as whether memory is shared or not. Where relevant, footnotes will be used to discuss the effect of implementation decisions.

Note that even though $defn_P$ is called in different environments during the parallel execution of the goals, it is still assumed that the new variables introduced 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.[1]

Two main problems were detected with this execution model.[2] The first one, related to the *variable binding conflict* of Conery [1983], appears whenever during the parallel execution of $\langle g_1, \theta\rangle$ and $\langle g_2, \theta\rangle$ the same variable is attempted to be bound to inconsistent values. Then, due to the standard definition of composition of substitutions (based on function composition) given in Lloyd [1987], Apt and van Emden [1982], and Apt [1990] the answers obtained by the parallel execution can be different from those obtained by the sequential execution, thus affecting the correctness of the model, as shown in Hermenegildo and Rossi [1995].

Example 2.1. Consider the state $\langle \texttt{p(x)} : \texttt{q(x)}, \epsilon\rangle$ and the following program:

$$\begin{array}{lll} \texttt{p(x)} & \leftarrow & \texttt{x} = \texttt{a}. \\ \texttt{q(x)} & \leftarrow & \texttt{x} = \texttt{b}. \end{array}$$

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

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, as pointed out in Hermenegildo and Rossi [1995].

[1] As defined, the execution model only finds the first answer to the goals. Several approaches to backtracking are possible. One is to avoid backtracking by computing in parallel all solutions to $\langle g'_1, \epsilon\rangle$ and $\langle g'_2, \epsilon\rangle$, storing them, and then (upon request) providing them in the appropriate order. However, in most implemented and-parallel systems, initially only the first solution to $\langle g'_1, \epsilon\rangle$ and $\langle g'_2, \epsilon\rangle$ is computed in parallel. If failure occurs later during the execution of $\langle G, \theta\theta_3\rangle$ and it reaches goal g_2, backtracking over g_2 is performed as in the sequential model. Only when backtracking reaches g_1, can this work be again performed in parallel with that of solving g_2. For generality, we will assume the second approach.

[2] A third problem was also detected in Hermenegildo and Rossi [1995] 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; thus, the no slow-down property may not hold. However, this problem was solved outside the scope of the theoretical model 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 this processor has a higher priority than those executing goals to the right.

Example 2.2. Consider the state $\langle \mathtt{p(x)} : \mathtt{q(x)}, \epsilon\rangle$ and the following program:

```
p(x)  ←  x = a.
q(x)  ←  x = b, proc, x = c.
```

where `proc` is very costly to execute. Both the sequential and parallel execution will fail, but their efficiency is quite different. While the sequential execution fails before executing `proc`, the parallel execution will first execute `proc` and then fail. △

The first solution proposed to solve these two problems was to only allow goals to be run in parallel if they do not share variables with respect to the current substitution [Conery 1983]. This was formally defined in Hermenegildo and Rossi [1995] as follows (and called "strict independence"):

Definition 2.3 [Hermenegildo and Rossi 1995]. Two goals g_1 and g_2 are said to be *strictly independent* with respect to a given substitution θ iff

$$vars(g_1\theta) \cap vars(g_2\theta) = \emptyset.$$

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 each $\theta \in \Theta$. Finally, a collection of goals is said to be simply strictly independent iff they are strictly independent for the set of all possible substitutions. △

The same definition can be applied to terms without any change. The authors of Hermenegildo and Rossi [1995] 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, significantly limiting the number of goals that may be executed in parallel. However, as pointed out in Hermenegildo and Rossi [1995], it has a very useful 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 strict independence, there have been several attempts to identify more general sufficient conditions. 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 DeGroot [1984], Warren et al. [1988], and Winsborough and Waern [1988], refined and formally defined in Hermenegildo and Rossi [1995] as follows:

Definition 2.4 [Hermenegildo and Rossi 1995]. A binding x/t is called a *v-binding* if t is a variable, otherwise it is called an *nv-binding*.△

Definition 2.5 [Hermenegildo and Rossi 1995]. 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 each $g_i\theta$ (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.$\triangle$

Intuitively, the first condition above 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 2.6. Consider the state $\langle \mathtt{p(x,y)} : \mathtt{q(y)}, \epsilon\rangle$ and the program:

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

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 might 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 the final substitution. This answer is obviously different from the θ obtained by the sequential execution, and so is an incorrect result. $\triangle$

As noticed in Hermenegildo and Rossi [1995], this could be solved by defining a "parallel composition" which avoids these problems. Since there is a natural bijection between substitutions and sets of equations in solved form, such parallel composition was defined in terms of "solving" the equations associated with the substitutions being composed. However, at that time adopting a new definition of composition would have 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 equations to reestablish the lost links (for a formal definition see Hermenegildo and Rossi [1995]).

Example 2.7. Consider the collection of goals $(r(x,z,x)$, $s(x,w,z)$, $p(x,y)$, $q(y))$ in some state (we consider θ already applied to the goals). According to the renaming transformation definition, we rewrite this to

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

Note that the first goal always remains unchanged. Equations of the form $x = x'$ above were called "back-bindings" (denoted by BB) and are related to the back-unification goals defined in Kalé [1987], and the closed environment concept of Conery [1987]. 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. 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 a new infinite derivation in the execution of $\langle g_2', \epsilon\rangle$ would yield a change. 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.

Possible inefficiency was 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, this 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" [Hermenegildo and Rossi 1995]. This concept was later (in some sense erroneously) identified with the preservation of the number of non failure nodes in the respective derivation trees. The second source of inefficiency was a failure when executing the back-bindings, 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 Hermenegildo and Rossi [1995], this does not ensure the preservation of the amount of work in failed derivations.

It is clear from the above discussion that the work developed in Hermenegildo and Rossi [1995] provided the basic results for logic programming. However, the definitions and proofs used are quite complex due to the introduction of the renaming transformation. In the next section we will generalize independence, search space preservation, and the parallel execution model to the constraint logic programming context. Somewhat surprisingly, we shall see that our generalization provides a more intuitive formalization of independence in the logic programming setting. In particular we will avoid using the renaming transformation, and we will be able to prove that the independence notions are not only sufficient but also necessary for ensuring correctness and efficiency.

3 A PARALLEL EXECUTION MODEL FOR CONSTRAINT LOGIC PROGRAMS

In this section we generalize the standard logic programming parallel execution model to the more general context of constraint logic programming (CLP), clarify what it means for parallel execution of goals to be correct and efficient with respect to the standard sequential evaluation of CLP, and formalize the concept of search space preservation as a necessary and sufficient condition for correctness and efficiency.

3.1 CLP Operational Semantics

First we revise the CLP scheme and the standard CLP operational semantics. In doing this, we will follow mainly Jaffar and Lassez [1987] and Jaffar and Maher [1994]. The interested reader should consult Jaffar and Maher [1994] for a more formal and more detailed account, as well as for the assumptions that are usually made about the constraint domain.

A *primitive constraint* has the form $p(\bar{t})$ where $\bar{t}$ is a sequence of arguments and p is a constraint predicate symbol. A *constraint* is a conjunction of primitive constraints. The empty constraint is denoted ϵ. A *literal* is an atom or a primitive constraint. The definitions of atom, rule, goal, and program are the natural generalization of those given earlier for logic programs.

CLP languages are parametrized by the allowed constants, functions, and constraint predicate symbols. These, together with their interpretation, constitute the underlying *constraint domain*. For example, standard Prolog can be viewed as a CLP language in which term equations, interpreted over the finite trees, form the constraint domain. As another example, the CLP language CLP($\Re$) [Jaffar and Michaylov 1987] extends Prolog by also providing the standard arithmetic constraints interpreted over the real numbers.

Let $\exists_{-\bar{x}}\phi$ denote the existential closure of the formula ϕ except for the variables $\bar{x}$ and $\tilde{\exists}\phi$ denote the full existential closure of ϕ.

The operational semantics is parametric in the *constraint solving* function, *consistent*, which tests the consistency of a constraint. That is, it returns *true* if the constraint is satisfiable, and *false* otherwise. For simplicity, we have assumed that the consistency test implemented by the constraint solver is complete. This allows us to treat constraints as logical formulae, and thus relate them by implication, logical equivalence, etc. However, our results continue to hold for incomplete solvers. In this case we just consider constraints as sets of (possibly delayed) primitive constraints and substitute conjunction by union, logical equivalence by syntactic equivalence, and implication by the subset relationship.

The operational semantics for CLP is very similar to that given earlier for logic programs. The main difference is that the substitution is replaced by a *constraint store* which collects the primitive constraints encountered so far, and the call to *mgu* is replaced by a call to the constraint solving function.

The operational semantics is therefore a transition system on states of the form $\langle G, c\rangle$ where G is a sequence of literals, and c is the constraint store. As before we also allow the state *fail*. Let a denote an atom and c' a constraint. The transition rules are

- $\langle a : G, c\rangle \rightarrow_r \langle B :: G, c\rangle$ if $B \in \mathit{defn}_P(a)$;
- $\langle a : G, c\rangle \rightarrow_{rf} \mathit{fail}$ if $\mathit{defn}_P(a) = \emptyset$;
- $\langle c' : G, c\rangle \rightarrow_c \langle G, c \wedge c'\rangle$ if $\mathit{consistent}(c \wedge c')$ holds;
- $\langle c' : G, c\rangle \rightarrow_{cf} \mathit{fail}$ if $\mathit{consistent}(c \wedge c')$ does not hold.

The definition of derivations, final states, successful and failed derivations, derivation trees, and success, infinite, and failure branches is a straightforward modification of those for logic programs. The constraint c is said to be a *partial answer* to state s if there is a derivation from s to a state $\langle G, c\rangle$, and it is said to be an *answer* if $\langle G, c\rangle$ is a final state (i.e., $G \equiv nil$). We denote the set of answers to state s for program P by $\mathit{ans}_P(s)$ and the partial answers by $\mathit{pans}_P(s)$.

3.2 A Model for the Parallel Execution of CLP

We will primarily be concerned with investigating independence from the viewpoint of parallelization. A necessary first step, therefore, is to generalize the parallel execution model given earlier for logic programs to CLP. Assume that we are given the state $\langle g_1 : g_2 : G, c\rangle$ and wish to execute g_1 and g_2 in parallel (the extension to more than two goals is straightforward). Our execution scheme is the following:[3]

- 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 $\langle G, c_s\rangle$.

Note that our parallel execution model is also intended to be generic, abstracting away from implementation details. We will again use footnotes to discuss the effect of implementation decisions. Also as before, we assume that the new variables introduced by defn_P during the parallel execution of the goals belong to disjoint sets.

The main difference between the parallel framework for LP and ours is that we replace substitution composition by conjunction. Indeed constraint conjunction corresponds exactly with the "parallel composition" needed in Hermenegildo and Rossi [1995]. What in the logic programming context would imply a reconsideration of the standard theory and results comes essentially for free with CLP. Therefore, we can avoid the need for the renaming transformation.

We must now formally define what it means for the parallel model to be correct and efficient with respect to the sequential one. It is easy to see that the only difference between these two models is that in the sequential model g_2 is executed with the constraint store c_1 corresponding to some answer to $\langle g_1, c\rangle$, while in the parallel model g_2 is executed with the constraint store c. Thus, we can base correctness and efficiency on the relationship between the execution of states $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$, for each c_1 computed.

[3] The subscript "s" will be associated to the arguments of the states obtained during the sequential execution. The subscript "r" will be associated to the arguments of the states obtained during the parallel execution of g_2 (the goal to the *r*ight).

The obvious definition of correctness, corresponding to that used for logic programming, is that execution of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ give rise to equivalent sets of answers.

Definition 3.1. Let s be the state $\langle g_1 : g_2 : G, c\rangle$ and P be a program. The parallel execution of g_1 and g_2 is *correct* iff for every $c_1 \in ans_P(\langle g_1, c\rangle)$ there exists a renaming ρ such that $\rho(s) \equiv s$, and a bijection which assigns to each $c_s \in ans_P(\langle g_2, c_1\rangle)$ an answer $c_r \in \rho(ans_P(\langle g_2, c\rangle))$ with $c_s \leftrightarrow (c_1 \wedge c_r)$. $\triangle$

However, this notion of correctness has two weaknesses. First, it does not ensure that answers are returned in the same order. This is desirable when parallelizing a program, since it guarantees that the order intended by the programmer is preserved. Second, it does not capture that successful derivations to the right of an infinite branch will never be explored. Thus we will also consider a more "operational" view of correctness.

Let $optree_P(s)$ be the tree obtained from the derivation tree of s by removing all nodes to the right of the first infinite branch in the tree, and let $opans_P(s)$ be the sequence of answers obtained in the in-order traversal of $optree_P(s)$.

Definition 3.2. Let s be the state $\langle g_1 : g_2 : G, c\rangle$ and P be a program. The parallel execution of g_1 and g_2 is *operationally correct* iff for every $c_1 \in ans_P(\langle g_1, c\rangle)$, the sequences $opans_P(\langle g_2, c_1\rangle)$ and $opans_P(\langle g_2, c\rangle)$ have the same length and there exists a renaming ρ such that $\rho(s) \equiv s$, and for all i, if c_s is the ith answer in $opans_P(\langle g_2, c_1\rangle)$ and c_r is the ith answer in $\rho(opans_P(\langle g_2, c\rangle))$, then $c_s \leftrightarrow (c_1 \wedge c_r)$. $\triangle$

Efficiency only requires that, in absence of failure (i.e., when g_1 has at least one answer), the amount of work performed by the second goal g_2 in the parallel model is less than or equal to that performed in the sequential model. We will not take into account the amount of work performed in conjoining the answers obtained from the parallel execution, since the cost of this is considered to be one of the overheads associated with the parallel execution (as creation of processes or tasks, scheduling, etc.).[4]

Also, we will assume for the moment that the cost of the application of each transition rule is constant and independent of the type of transition 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 3.3. The *cost* of evaluating state s, written $cost(s)$, is

$$\sum_{i \in TR} K_i * N(i, s).$$

Definition 3.4. Let $\langle g_1 : g_2 : G, c\rangle$ be a state and P be a program. The parallel

[4] And, in fact, in shared memory machines this step is performed on the fly, at minimal cost, since the goals are generally run in a shared environment.

execution of g_1 and g_2 is *efficient* iff for every $c_1 \in ans_P(\langle g_1, c\rangle)$,[5]

$$cost(\langle g_2, c\rangle) \leq cost(\langle g_2, c_1\rangle).\triangle$$

4 SEARCH SPACE PRESERVATION

We will now identify independence conditions for goals which will ensure that parallel execution of the goals is correct and efficient. As a first step in this quest to identify independence conditions, we shall formalize search space preservation and clarify its relationship with correctness and efficiency of the parallel execution. Search space preservation allows us to understand correctness and efficiency in terms of derivation trees.

We assume that nodes in a derivation tree are labeled with their path, i.e., they are labeled with a unique identifier obtained by concatenating the relative position of the node among its siblings to the path of the parent node. We also assume that some pre-defined order is assigned to the bodies in $defn_P(a)$, and that this order is inherited by the associated child nodes.

Definition 4.1. Two nodes n and n' in the derivation trees of states s and s', respectively, with the same path *correspond* iff 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 transition rules.$\triangle$

Definition 4.2. 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 $tree_P(s')$. They have the *same operational search space* for program P iff there exists a (total) bijection which assigns to each node in $optree_P(s)$ its corresponding node in $optree_P(s')$.$\triangle$

We first show that search space preservation is sufficient for ensuring correctness and efficiency. That is to say, that given a state $\langle g_1 : g_2 : G, c\rangle$ and a program P, the parallel execution of g_1 and g_2 is correct and efficient if, for every $c_1 \in ans_P(\langle g_1, c\rangle)$, the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same as for P. Ensuring efficiency is straightforward due to the definition of search space, which provides a bijection among the same transitions. The proof of correctness is a little more complex and relies on the following two lemmas which relate the derivation trees for states with the same goal but with different constraints.

Intuitively, the following lemma guarantees that for states with the same goal, and in the absence of failure, the goals associated to nodes with the same path in different derivation trees (possibly starting from different initial constraints) must be identical up to renaming.

---

[5]If we consider a model in which, during the parallel execution, all solutions to the parallel goals are computed, the condition above can be relaxed: we can just require the cost of executing $\langle g_2, c\rangle$ multiplied by the number of answers in $ans_P(\langle g_1, c\rangle)$, to be less than or equal to the sum of the cost of executing $\langle g_2, c_1\rangle$, for each answer $c_1 \in ans_P(\langle g_1, c\rangle)$. Furthermore, if after the parallel execution the parallel goals share their environments, the above definition could have been specialized so that only the amount of work up to the first solution for $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ (for each c_1) is taken into account, since the rest are explored in the same environment as the sequential one.

Lemma 4.3. Let $\langle G, c_1\rangle$ and $\langle G, c_2\rangle$ be two states and P a program. There exists a renaming γ such that

- for every two non failure nodes $\langle G_1', c_1'\rangle$ and $\langle G_2', c_2'\rangle$ with the same path in $tree_P(\langle G, c_1\rangle)$ and $tree_P(\langle G, c_2\rangle)$, respectively, $G_1' \equiv \gamma(G_2')$;
- $\gamma(G) = G$, $\gamma(c_1) = c_1$ and $\gamma(c_2) = c_2$;
- γ is its own inverse.$\triangle$

PROOF. The proof is by induction over the non failure nodes $n_0, ..., n_k$ in the tree $tree_P(\langle G_1, c_1\rangle)$ such that for each of these nodes, n_i say, there is a non failure node n_i' in $tree_P(\langle G, c_1\rangle)$ with the same path as n_i. Without loss of generality, we can assume that all nodes come after their parent in the sequence. We shall inductively define renamings $\gamma_0, ..., \gamma_k$ such that γ_i satisfies the two conditions in the lemma statement for nodes $n_0, ..., n_i$ respectively.

The base case for the induction argument is for n_0. As a result, n_0 and n_0' must be the roots of their trees, i.e., $n_0 \equiv \langle G, c_1\rangle$ and $n_0' \equiv \langle G, c_2\rangle$. Choosing γ_0 to be the identity renaming clearly satisfies the induction hypothesis.

Now, consider the nodes $n_k \equiv \langle G_k, c_k\rangle$ and $n_k' \equiv \langle G_k', c_k'\rangle$. Since the parent of n_k and that of n_k' must be non failure nodes (from the definition of the operational semantics only non failure nodes can have children) and must have the same path, they occur in the induction sequence of nodes. Let their parents be $n_p \equiv \langle G_p, c_p\rangle$ and $n_p' \equiv \langle G_p', c_p'\rangle$, respectively. Now, since $p < k$, we have from the induction hypothesis that

$$G_p \equiv \gamma_{k-1}(G_p'). \qquad (1)$$

By assumption, n_k and n_k' are non failure nodes, and by definition of the operational semantics, non failure nodes can only be obtained by a $\rightarrow_c$ transition or by a $\rightarrow_r$ transition.

If n_k was obtained by a $\rightarrow_c$ transition, then the leftmost literal in G_p is a constraint. From (1), the leftmost literal in G_p' must also be a constraint, and therefore G_k' was obtained from G_p' also using a $\rightarrow_c$ transition. From the definition of the $\rightarrow_r$ transition and (1), it follows that $G_k \equiv \gamma_{k-1}(G_k')$. Thus, we can choose γ_k to be γ_{k-1}.

On the other hand, if n_k was obtained by a $\rightarrow_r$ transition, then the leftmost literal in G_p is an atom, say h. Analogously to above, from (1), the leftmost literal in G_p' must also be a variant of h, say h', and so G_k' was obtained from G_p' also using a $\rightarrow_c$ transition. Since rules are applied in order and the nodes n_k and n_K' have the same path, G_k and G_k' must have been obtained using renamings ρ and ρ', respectively, of the same program rule $h_P \leftarrow B_P$. Define the renaming ρ_{local} by

$$\rho_{local}(x) = \begin{cases} \rho'(\rho^{-1}(x)) & \text{if } x \in vars(\rho(B_P)) \setminus vars(h) \\ \rho(\rho'^{-1}(x)) & \text{if } x \in vars(\rho'(B_P)) \setminus vars(h') \\ x & \text{otherwise.} \end{cases}$$

ρ_{local} maps each local variable in $\rho(B_P)$ to the corresponding local variable in $\rho'(B_P)$ and vice versa. Note that since $defn_P$ always renames local variables to distinct new variables, the local variables in $\rho'(B_P)$ and $\rho(B_P)$ are distinct and do not occur in nodes $n_1, ..., n_k$ or $n_1', ..., n_k'$. By construction, $\gamma_k = \gamma_{k-1} \circ \rho_{local}$ is a renaming. Furthermore, for $i = 1, ..., k-1$, $\gamma_k(n'_i) = \gamma_{k-1}(n'_i)$ and for n_k,

$$\gamma_k(G'_k) = \gamma_k(\rho'(B_P) : G'_p \setminus h') = \rho(B_P) : \gamma_{k-1}(G'_p \setminus h') = G_k$$

since for each local variable x in B_p,

$$\gamma_k(\rho'(x)) = \rho_{local}(\rho'(x)) = \rho(x)$$

and for each non local variable x in B_P,

$$\gamma_k(\rho'(x)) = \gamma_{k-1}(\rho'(x)) = \rho(x)$$

as $\gamma_{k-1}(h') = h$. By construction, ρ_{local} is its own inverse. Furthermore, $\gamma_k = \gamma_{k-1} \circ \rho_{local} = \rho_{local} \circ \gamma_{k-1}$, since ρ_{local} and γ_{k-1} only affect disjoint sets of variables. It follows that γ_k is its own inverse Thus, γ_k satisfies the induction argument. $\square$

We note that the first condition of Lemma 4.3 can also be equivalently expressed as: for every two non failure nodes $\langle G'_1, c'_1\rangle$ and $\langle G'_2, c'_2\rangle$ with the same path in $tree_P(\langle G, c_1\rangle)$ and $\gamma(tree_P(\langle G, c_2\rangle))$, respectively, $G'_1 \equiv G'_2$. We will make use of this alternative formulation when convenient.

The renaming γ constructed in the proof of the preceding lemma allows us to map nodes from $tree_P(\langle G, c_2\rangle)$ to $tree_P(\langle G, c_1\rangle)$ by taking into account the effect of local variable renamings performed in the operational semantics with calls to $defn_P$. We call γ the *local variable correcting renaming* for $tree_P(\langle G, c_1\rangle)$ and $tree_P(\langle G, c_2\rangle)$.

When focusing on parallelism, the above lemma guarantees that, in absence of failure, the goals associated with every two nodes with the same path in the parallel and sequential execution, respectively, are identical up to renaming by the local variable correcting renaming γ. As a result, it is easy to prove the following lemma which shows that for non failure nodes the constraint obtained during the sequential execution (c_s) is equivalent to the conjunction of the constraints obtained during the parallel executions (c_1 and c_r).

Lemma 4.4. Let $\langle g_2, c_1\rangle$ and $\langle g_2, c\rangle$ be two states with $c_1 \rightarrow c$ and P be a program. 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 $\gamma(tree_P(\langle g_2, c\rangle))$, respectively, $c_s \leftrightarrow (c_1 \wedge c_r)$, where γ is the the local variable correcting renaming for $tree_P(\langle g_2, c_1\rangle)$ and $tree_P(\langle g_2, c_2\rangle)$.$\triangle$

Proof. By definition of the operational semantics, all parent nodes of a given node are known to be non failure. By Lemma 4.3, the sequences of literals of all parents of s are identical to those of all parents of r with the same path. This means that the constraints added to c_1 and to c, yielding c_s and c_r respectively, have been the same. Since by assumption $c_1 \rightarrow c$, and therefore $c_1 \leftrightarrow c_1 \wedge c$, it is clear that $c_s \leftrightarrow c_1 \wedge c_r$. $\square$

The above lemma 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, and thus the following results:

Theorem 4.5. *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 if for every $c_1 \in ans_P(\langle g_1, c\rangle)$, the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same for P.*$\triangle$

PROOF. By definition of search space preservation, there exists a bijection which assigns to every final state $r \equiv \langle G_r, c_r \rangle$ in $tree_P(\langle g_2, c \rangle)$ a final state $s \equiv \langle G_s, c_s \rangle$ with the same path in $tree_P(\langle g_2, c_1 \rangle)$, thus establishing a bijection among the answers. By Lemma 4.3, there exists a renaming γ for initial states $\langle g_2, c_1 \rangle$ and $\langle g_2, c \rangle$ such that $G_s \equiv \gamma(G_r)$. Also, since $c_1 \in ans_P(\langle g_1, c \rangle)$ we know that $c_1 \rightarrow c$. Thus, by Lemma 4.4, $c_s \leftrightarrow c_1 \wedge \gamma(c_r)$, and we have proved correctness.

Let us now prove efficiency. By definition of search space preservation, there exists a bijection among every node in $tree_P(\langle g_2, c \rangle)$ and a node with the same path in $tree_P(\langle g_2, c_1 \rangle)$ which is obtained with the same transition rule. Thus, for each $i \in TR : N(i, \langle g_2, c \rangle) = N(i, \langle g_2, c_1 \rangle)$. As a result, for every c_1: $\sum_{i \in TR} K_i * N(i, \langle g_2, c \rangle) = \sum_{i \in TR} K_i * N(i, \langle g_2, c_1 \rangle)$, i.e., $cost(\langle g_2, c \rangle) = cost(\langle g_2, c_1 \rangle)$. We have thus proved efficiency. □

Using a similar proof it is straightforward to show that:

THEOREM 4.6. *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 operationally correct and efficient if for every $c_1 \in ans_P(\langle g_1, c \rangle)$, the operational search spaces of $\langle g_2, c \rangle$ and $\langle g_2, c_1 \rangle$ are the same for P.*△

It is easy to see that search space preservation is not necessary for ensuring correctness, since correctness is not affected by search space changes in either failure or infinite branches. However, we can show that search space preservation is necessary for ensuring that both correctness *and* efficiency hold. The following two lemmas are instrumental in proving this, since they show 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 if a branch in $tree_P(\langle g_2, c \rangle)$ does not appear in $tree_P(\langle g_2, c_1 \rangle)$.

Lemma 4.7. Let $\langle g_2, c_1 \rangle$ and $\langle g_2, c \rangle$ be two states 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.△

PROOF. Let us first assume that s and r have been obtained by the same transition rule. Then, by definition of the operational semantics, either both are identical to $fail$ or both are non failure. For proving the other direction let s' and r' be the parents of s and r, respectively. By definition of the operational semantics we know that s' and r' are non failure nodes. Thus, by Lemma 4.3, if the leftmost literal in the sequence of literals in s' is an atom (resp. constraint) then the leftmost literal in the sequence of literals in r' must be a variant of the same atom (resp. constraint). Then, by definition of the operational semantics, if $s \equiv r \equiv fail$ they must have been obtained by applying $\rightarrow_{rf}$ (leftmost literal is an atom) or $\rightarrow_{cf}$ (leftmost literal is a constraint), and if they are both non failure nodes, they must have been obtained by applying $\rightarrow_r$ (leftmost literal is an atom) or $\rightarrow_c$ (leftmost literal is a constraint). □

Lemma 4.8. Let $\langle g_2, c_1 \rangle$ and $\langle g_2, c \rangle$ be two states 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.$\triangle$

PROOF. Let us assume that s is the first node in its branch for which there is no corresponding node. Let s' be the parent of s. By definition of the operational semantics s' is non failure, say $s' \equiv \langle G'_s, c'_s \rangle$. By assumption, s' has a corresponding node r' with the same path in $tree_P(\langle g_2, c\rangle)$. By Lemma 4.7, r' must also be non failure, say $r' \equiv \langle G'_r, c'_r \rangle$. By Lemma 4.3, $G'_s \equiv \gamma(G'_r)$ where γ is the the local variable correcting renaming for $tree_P(\langle g_2, c_1\rangle)$ and $tree_P(\langle g_2, c\rangle)$. We note that the first literal in G'_s cannot be an atom. If so, s must have been obtained by applying either $\rightarrow_r$ or $\rightarrow_{rf}$ and so r' must have a child r obtained using $\rightarrow_r$ or $\rightarrow_{rf}$, respectively. This would mean that r corresponds to s contradicting the assumption that s has no corresponding node. As the first literal in G'_s must be a constraint, r' has a single child r obtained by using either $\rightarrow_c$ or $\rightarrow_{cf}$. Clearly r has the same path as s. Now if s has been obtained by applying the $\rightarrow_c$ transition rule, then $consistent(c'_s \wedge c')$ must hold. However, by Lemma 4.4 $c'_s \leftrightarrow c'_1 \wedge \gamma(c'_r)$. Therefore, $consistent(c' \wedge \gamma(c'_r))$ must also hold, and thus r must also be obtained by applying the $\rightarrow_c$ rule. But this contradicts the assumption that r and s do not correspond. The only possibility is that while $consistent(c'_s \wedge c')$ does not hold, $consistent(c' \wedge \gamma(c'_r))$ holds. Thus, s and r must have been obtained by applying the $\rightarrow_{cf}$ and $\rightarrow_c$ transition rules, respectively. $\square$

It follows from the above lemmas that, for each two states $\langle g_2, c_1\rangle$ and $\langle g_2, c\rangle$ such that $c_1 \rightarrow c$, all non failure nodes in the tree of $\langle g_2, c_1\rangle$ correspond with the nodes with the same path in the tree of $\langle g_2, c\rangle$. Failure nodes will also correspond unless a longer branch is obtained in $\langle g_2, c\rangle$ due to the less constrained store. However, the assumption of efficiency ensures that such long branches do not exist. Thus, search space preservation is necessary to ensure efficiency and the following theorems hold:

THEOREM 4.9. *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 $c_1 \in ans_P(\langle g_1, c\rangle)$ the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same for P.*$\triangle$

PROOF. Since we have already proved that search space preservation is sufficient, let us focus on the necessary condition. Let us reason by contradiction and assume that the parallel execution is correct and efficient but there exists at least one $c_1 \in ans_P(\langle g_1, c\rangle)$ for which the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are not the same for P. By Lemma 4.8 we know that for every node in $tree_P(\langle g_2, c_1\rangle)$ obtained with one of the transition rules in $\{\rightarrow_r, \rightarrow_c, \rightarrow_{rf}\}$, there exists a corresponding node in $tree_P(\langle g_2, c\rangle)$ which has been obtained with the same transition rule. Thus, for every $i \in \{\rightarrow_r, \rightarrow_c, \rightarrow_{rf}\}$: $N(i, \langle g_2, c_1\rangle) \leq N(i, \langle g_2, c\rangle)$. Also, for every node s in $tree_P(\langle g_2, c_1\rangle)$ obtained with the $\rightarrow_{cf}$ transition rule, either it has a corresponding node in $tree_P(\langle g_2, c\rangle)$ or, by Lemma 4.8 there exists a node r in $tree_P(\langle g_2, c_1\rangle)$ with the same path, which have been obtained applying the $\rightarrow_c$ transition rule. By definition of correctness there exists a bijection among answer nodes, i.e., nodes in successful derivations. Thus r must be non failure, and the branches starting at r must be either infinite or failure. Thus the amount of work performed for obtaining r and its children is greater than that performed for obtaining s. But then the parallel execution is not efficient, contradicting the initial assumption. $\square$

THEOREM 4.10. *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 operationally correct and efficient iff for every $c_1 \in ans_P(\langle g_1, c\rangle)$ the operational search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same for P.* $\triangle$

PROOF. Direct from Theorem 4.9 and the fact that the bijection is among nodes with the same path, thus providing the connection between the answers in the same position of the sequences. □

Note that theorems 4.9 and 4.10 imply that, in absence of failure, the amount of work performed during the parallel execution is equal to (and no less) than that performed in the sequential execution, with any possible speedup coming from the parallel execution of this work.

These results also allow us to clarify one of the points mentioned in Section 2. Let $\#nfnodes_P(s)$ be the number of non failure nodes in the derivation tree of state s for program P.

Corolary 4.11. Let $\langle g_2, c_1\rangle$ and $\langle g_2, c\rangle$ be two states such that $c_1 \rightarrow c$. The search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same for program P iff $\#nfnodes_P(\langle g_2, c\rangle) = \#nfnodes_P(\langle g_2, c_1\rangle)$.$\triangle$

PROOF. Let us first assume that the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same for P. From the definition of search space preservation, there exists a bijection among nodes and, thus, $\#nfnodes_P(\langle g_2, c\rangle) = \#nfnodes_P(\langle g_2, c_1\rangle)$. For proving the other direction let us reason by contradiction and assume that $\#nfnodes_P(\langle g_2, c\rangle) = \#nfnodes_P(\langle g_2, c_1\rangle)$ but the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are different for P. By Lemma 4.8, every non corresponding node s in $\langle g_2, c_1\rangle$ must be a failure node, and the node r with the same path in $\langle g_2, c\rangle$ must be a non failure node. But this implies that $\#nfnodes_P(\langle g_2, c\rangle) > \#nfnodes_P(\langle g_2, c_1\rangle)$, which contradicts the initial assumption. □

This justifies why preservation of search space was identified with preservation of the number of non failure nodes in the logic programming context. However, as we will see in Section 8, this identification cannot be performed when co-routining is provided, since then a more constrained store c_1 can both prune and *enlarge* the search space.

We have now proven 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 may 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 Hermenegildo and Rossi [1995], 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. We will return to this issue in Section 7.

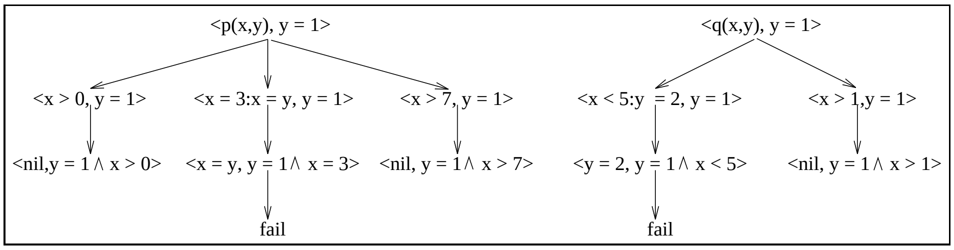

Fig. 1.

5 LEVELS OF INDEPENDENCE

In this section we will investigate various levels of independence for CLP languages. As discussed in Section 2, independence was also defined at several levels of strength for logic programs, starting from very restrictive definitions which ensure search space preservation, and then relaxing these conditions in order to enlarge the number of goals which can be considered independent, while still preserving the search space. We will do the opposite: start from a weak notion of independence which is not sufficient for ensuring search space preservation, and then progressively restrict this definition until it does imply search space preservation. This allows us to systematically discuss and compare these different notions and their applications: although parallel execution is arguably the most important application of independence, it is not the only one.

5.1 Weak Independence

The first level is a relatively lax notion of independence which captures the intuitive idea that simply guaranteeing "consistency among answers" of goals is sufficient for the purposes of a number of applications.

Example 5.1. Consider the following fragment of a CLP($\Re$) program:

```
p(x,y):- x > 0.                 q(x,y):- x < 5 , y = 2.
p(x,y):- x = 3, x = y.          q(x,y):- x > 1.
p(x,y):- x > 7.
```

Figure 1 shows each possible derivation for states $\langle \texttt{p(x,y)}, c\rangle$ and $\langle \texttt{q(x,y)}, c\rangle$ where c is $y = 1$. Since the answers of $\langle \texttt{p(x,y)}, c\rangle$ are consistent with those of $\langle \texttt{q(x,y)}, c\rangle$ then `p(x,y)` and `q(x,y)` can be considered in some sense independent for c.$\triangle$

Let us now formally define this level of independence which we will call *weak independence*:

Definition 5.2. Goals g_1 and g_2 are *weakly independent* for constraint c and program P iff

$$\forall c_1 \in ans_P(\langle g_1, c\rangle) \text{ and } \forall c_r \in ans_P(\langle g_2, c\rangle), consistent(c_1 \wedge c_r).$$

A collection of goals $g_1 : \cdots : g_n$ is weakly independent for a given c 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 and P. $\triangle$

Note that, according to this definition, goals which fail (those for which the set of answers is empty) for a given constraint are weakly independent of all other goals. Also note that the appropriateness of the definition depends on the assumption that $defn_P$ renames local variables in $ans_P(\langle g_1, c\rangle)$ and $ans_P(\langle g_2, c\rangle)$ apart. Without this assumption, they would need to be existentially quantified.This is also true for subsequent definitions of independence.

Lemma 5.3. Goals g_1 and g_2 are weakly independent for constraint c and program P iff $\forall c_1 \in ans_P(\langle g_1, c\rangle)$, there exists a bijection which assigns to each node in a successful branch of $tree_P(\langle g_2, c\rangle)$ a corresponding node in a successful branch of $tree_P(\langle g_2, c_1\rangle)$.$\triangle$

PROOF. Let γ be the local variable correcting renaming for $tree_P(\langle g_2, c\rangle)$ and $tree_P(\langle g_2, c_1\rangle)$.

Let us first assume that, for each $c_1 \in ans_P(\langle g_1, c\rangle)$, there exists a bijection which assigns to each node in a successful branch of $tree_P(\langle g_2, c\rangle)$ a corresponding node in a successful branch of $tree_P(\langle g_2, c_1\rangle)$. Consider some answer $c_1 \in ans_P(\langle g_1, c\rangle)$ and answer $c_r \in ans_P(\langle g_2, c\rangle)$. Since c_r is an answer, there is a success node $r \equiv \langle nil, c_r\rangle$ in $tree_P(\langle g_2, c\rangle)$. By assumption, there is a corresponding node s in $tree_P(\langle g_2, c_1\rangle)$. From Lemma 4.3, $s \equiv \langle nil, c_s\rangle$. From Lemma 4.4 we have that c_s is equivalent to $(c_1 \wedge \gamma(c_r))$. Since s is on a successful branch, c_s is consistent. Thus $\gamma(c_s)$ is consistent. From Lemma 4.3, $\gamma(c_1) = c_1$ and γ is its own inverse. Thus,

$$\gamma(c_s) \equiv \gamma(c_1 \wedge \gamma(c_r)) \equiv \gamma(c_1) \wedge \gamma(\gamma(c_r))) \equiv c_1 \wedge c_r$$

and so $consistent(c_1 \wedge c_r)$ holds.

Now consider the other direction. Let us assume that g_1 and g_2 are weakly independent for c and P. By Lemma 4.8, for all non failure nodes, and in particular those in successful branches of $tree_P(\langle g_2, c_1\rangle)$, there exists a corresponding node with the same path in a successful branch of $tree_P(\langle g_2, c\rangle)$. From the assumption of weak independence, for each node $r \equiv \langle G_r, c_r\rangle$ in a successful branch of $tree_P(\langle g_2, c\rangle)$ we have that $consistent(c_1 \wedge c_r)$ holds. And since

$$\gamma(c_1 \wedge c_r) \equiv \gamma(c_1) \wedge \gamma(c_r) \equiv c_1 \wedge \gamma(c_r)$$

we have that $consistent(c_1 \wedge \gamma(c_r))$ must also hold. By Lemma 4.4, the consistency tests for obtaining the nodes with the same path as r are performed over a constraint c_s satisfying $c_s \leftrightarrow c_1 \wedge \gamma(c_r)$, and thus $consistent(c_s)$ must also hold. As a result, there exists a non failure node s in $tree_P(\langle g_2, c_1\rangle)$ with the same path as r. And by Lemma 4.7 we have that s and r correspond. $\square$

The usefulness of weak independence is based on the following result:

THEOREM 5.4. *Let $g_1 : \cdots : g_n$ be a collection of weakly independent goals for constraint c and program P. Let $g_i, 1 \leq i \leq n$ be a goal such that there exists $c_1 \in ans_P(\langle g_1 : \cdots : g_{i-1}, c\rangle)$ with $ans_P(\langle g_i, c_1\rangle) = \emptyset$. Then, for every $c_2 \in ans_P(\langle g_1 : \cdots : g_{i-1}, c\rangle)$, $ans_P(\langle g_i, c_2\rangle) = \emptyset$.*$\triangle$

PROOF. By assumption, the collection of goals $g_1 : \cdots : g_i$ is weakly independent for c and P. By definition of weak independence, g_i and the goal $g_1 : \cdots : g_{i-1}$ are weakly independent for c and P. Also by assumption we have that there exists $c_1 \in ans_P(\langle g_1 : \cdots : g_{i-1}, c\rangle)$ with $ans_P(\langle g_i, c_1\rangle) = \emptyset$. This means that there is no successful branch in $tree_P(\langle g_i, c_1\rangle)$. By Lemma 5.3 we have that $\forall c_1 \in ans_P(\langle g_1 : \cdots : g_{i-1}, c\rangle)$, there exists a bijection which assigns to each node in a successful branch of $tree_P(\langle g_i, c\rangle)$ a corresponding node in a successful branch of $tree_P(\langle g_i, c_1\rangle)$. As a result, there can be no successful branches in $tree_P(\langle g_i, c\rangle)$. From the definition of the operational semantics, for every $c_2 \in ans_P(\langle g_1 : \cdots : g_{i-1}, c\rangle)$ we have that $c_2 \rightarrow c$. By completeness of *consistent*, there can be no successful branches in any $tree_P(\langle g_i, c_2\rangle)$ such that $c_2 \rightarrow c$. Thus, $ans_P(\langle g_i, c_2\rangle) = \emptyset$. □

The above property is, in principle, useful for performing optimizations which are based on determination of producer-consumer relationships, such as intelligent backtracking. Backtracking occurs during exploration of the derivation tree whenever a failure node is reached. In the standard operational semantics, control "backtracks" to the closest ancestor with unexplored branches, thus ensuring depth-first exploration of the derivation tree. With intelligent backtracking [Bruynooghe and Pereira 1984], however, control may directly backtrack further up the tree. It requires analyzing, upon failure, the causes of the failure and determining the appropriate ancestor to backtrack to that can eliminate the failure while maintaining correctness, thus avoiding unnecessary computation.

A simple form of intelligent backtracking can be based on the notion of weak independence. Let $g_1 : \cdots : g_n$ be a set of goals which are weakly independent for the store c. Theorem 5.4 ensures that whenever there exists a goal $g_i, 1 \leq i \leq n$ for which no answers for goal g_i are found, execution can safely backtrack to the choice-point placed just before g_1, skipping all the choice-points in between.

It follows from the results in the previous section, that weak independence is not sufficient for ensuring search space preservation, since only successful derivations of the goals have been considered and the search space can also be affected through interactions with derivations failed or infinite derivations.

Example 5.5. Consider the previous example. Assume that we start from the state $\langle \mathtt{p(x,y)} : \mathtt{q(x,y)}, \mathtt{y} = \mathtt{1}\rangle$. It is clear that the search space associated with $\langle \mathtt{q(x,y)}, \mathtt{y} = \mathtt{1} \wedge \mathtt{x} > \mathtt{7}\rangle$ is smaller than that associated with $\langle \mathtt{q(x,y)}, \mathtt{y} = \mathtt{1}\rangle$, since the derivation in which $x < 5$ appears would fail earlier – as soon as $\mathtt{x} < \mathtt{5}$ is checked for consistency with the store. △

5.2 Strong Independence

We can define a more restrictive concept of independence, in the spirit suggested above, by taking into account all partial answers:

Definition 5.6. Goal g_2 is *strongly independent* of goal g_1 for constraint c and program P iff

$$\forall c_1 \in ans_P(\langle g_1, c\rangle) \text{ and } \forall c_r \in pans_P(\langle g_2, c\rangle), consistent(c_1 \wedge c_r)$$

A collection of goals $g_1 : \cdots : g_n$ is strongly independent for a given c and P iff for every $g_i, 1 \leq i \leq n$, then g_i is strongly independent of the goal $g_1 : \cdots : g_{i-1}$ for c and P. △

Note that while weak independence is symmetric, strong independence is not.

Example 5.7. In the example given in Figure 1, `p(x,y)` is strongly independent of `q(x,y)` for the constraint $c \equiv \mathtt{y} = \mathtt{1}$, since all answers to $\langle \mathtt{q(x,y)}, c\rangle$ are consistent with partial answers of $\langle \mathtt{p(x,y)}, c\rangle$. However, `q(x,y)` is not strongly independent of `p(x,y)` for the same constraint c. $\triangle$

Also, note that if a goal g_2 is strongly independent of another goal g_1 for c, then g_1 and g_2 are weakly independent for c.

We will now prove some properties of strongly independent goals. The main result is that goal g_2 is independent of g_1 for a given constraint c if and only if the search space is preserved. Intuitively the following theorem states that consistency between the answers of $\langle g_1, c\rangle$ and the partial answers of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ precludes pruning any branches, thus ensuring search space preservation. And vice-versa, search space preservation indicates no pruning and, thus, consistency.

THEOREM 5.8. *Goal g_2 is strongly independent of goal g_1 for constraint c and program P iff*

$$\forall c_1 \in ans_P(\langle g_1, c\rangle), \textit{the search spaces of } \langle g_2, c\rangle \textit{ and } \langle g_2, c_1\rangle \textit{ are the same.} \triangle$$

PROOF. Let γ be the local variable correcting renaming for $tree_P(\langle g_2, c\rangle)$ and $tree_P(\langle g_2, c_1\rangle)$.

Let us first assume that $\forall c_1 \in ans_P(\langle g_1, c\rangle)$ the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are the same. By definition of search space, there exists a bijection which assigns to each node r in $tree_P(\langle g_2, c\rangle)$ a corresponding node s in $tree_P(\langle g_2, c_1\rangle)$. By Lemma 4.7 each r and s are either both failure or both non failure nodes. For non failure nodes, say $s \equiv \langle G_s, c_s\rangle$ and $r \equiv \langle G_r, c_r\rangle$, by Lemma 4.4 we have that c_s is consistent and equivalent to $(c_1 \wedge \gamma(c_r))$. From properties of γ it follows that $consistent(c_1 \wedge c_r)$ must also hold. Since r refers to the nodes not only in successful but also in failure branches, it contains all partial answers and, thus, g_2 is strongly independent of g_1 for c and P.

The other direction uses a proof by contradiction. Let us assume that while g_2 is strongly independent of g_1 for c and P, the search spaces of $\langle g_2, c\rangle$ and $\langle g_2, c_1\rangle$ are not the same. By Lemma 4.8, there must exist a node $s \equiv fail$ in $tree_P(\langle g_2, c_1\rangle)$ obtained by applying $\rightarrow_{cf}$ and a node $r \equiv \langle G_r, c_r\rangle$ in $tree_P(\langle g_2, c\rangle)$ with the same path obtained by applying $\rightarrow_c$ such that their parents correspond. By construction, $\gamma(c_1) \equiv c_1$, and thus, by strong independence, $consistent(c_1 \wedge \gamma(c_r))$ must also hold. By Lemma 4.4, the consistency test performed for obtaining s was applied to the constraint $c_s \leftrightarrow c_1 \wedge \gamma(c_r)$. But, $consistent(c_s)$ holds, contradicting s being $fail$. $\square$

This theorem ensures that strong independence is not only sufficient but also necessary for ensuring preservation of search space. Thus, from Theorems 4.9 and 4.10, correctness and efficiency of the parallel execution of a set of strongly independent goals for current constraint store c holds iff each goal is strongly independent for current constraint store c of the sequence composed of goals to its left.

Apart from parallelization, this theorem also provides a basis for goal reordering, an important optimization for CLP languages. Marriott and Stuckey [1992] suggested reordering the goal $c \wedge g$ to $g \wedge c$, where c is a primitive constraint, whenever c and g are strongly independent for all possible constraint stores occurring before executing c and g. The motivation for this is that variables in c may become uniquely defined by g, enabling the constraint c to be replaced by either an assignment statement or a simple Boolean test. If this is true, especially in the case g is recursive, large speedups are obtained. We can lift this idea to the level of goals and thus reorder goals as well.

It is difficult to give simple yet general conditions which ensure that the reordering of two goals reduces the search space. However, one simple condition that ensures that the reordering does not increase the search space is that the rightmost goal is "single solution" and strongly independent of the leftmost goal. Note that any deterministic goal, and in particular a primitive constraint, is single solution.

Definition 5.9. A goal g is *single solution* for constraint c and program P iff the state $\langle g, c\rangle$ has at most one successful derivation in P.

THEOREM 5.10. *If goal g_2 is both strongly independent of goal g_1 and single solution for constraint c and P then*

$$cost(\langle g_2 : g_1, c\rangle) \leq cost(\langle g_1 : g_2, c\rangle). \triangle$$

The proof comes directly from Lemma 4.8 and the given CLP operational semantics. Note that the search space can be decreased for two reasons. First, due to the asymmetry of strong independence g_2 can decrease the search space of g_1 for c. Second, the answer for g_2 (if any) will not be recomputed when each answer to g_1 is found.

5.3 Search Independence

As discussed in Section 2.2, in the independent and-parallel model, parallel goals are executed in different environments. The isolation of the environments quite accurately reflects the actual situation in distributed implementations of independent and-parallelism [Conery 1987]. However, in models designed for shared addressing space machines, such 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 (e.g., Hermenegildo and Greene [1990] and Lin [1988]). The amount of overhead introduced by requiring isolated environments (either copying the environment or renaming the goals, plus conjoining the solutions) in these machines, suggests that such isolation should not be implemented unnecessarily. Furthermore, sharing the environments allows us to avoid performing the conjunction of the answers obtained from the parallel execution, since this happens automatically through the use of a shared constraint store. One might think that if we ensure that g_2 is strongly independent of g_1 with respect to a given constraint store c, then we can execute them in parallel in the same environment while preserving the correctness and efficiency with respect to the sequential execution of $\langle g_1 : g_2, c\rangle$. However, this is not true since, again, we have only considered the successful derivations of g_1 and, in this new context, it is possible for the execution of $\langle g_2, c\rangle$ to prune the search space of $\langle g_1, c\rangle$, which may lead to incorrect answers.

Example 5.11. Consider the following CLP($\Re$) program:

```
p(x):- x = 2, fail.                q(x):- x = 1.
p(x):- x = 1.
```

Clearly `q(x)` is strongly independent of `p(x)` for *true*. Now consider the parallel execution of $\langle$`p(x)` : `q(x)`$, true\rangle$ in an environment with a shared constraint store. If parallel reduction starts by rewriting `p(x)` with the first rule, and the constraint `x = 2` is processed, then the store becomes `x = 2`. Now if `q(x)` is reduced and `x=1` is added to the store, failure will result and evaluation of `q(x)` will backtrack. As there is no other rule in the definition of `q(x)`, evaluation will wrongly fail, without finding the answer. $\triangle$

For this reason we define a symmetric notion of strong independence which ensures that neither goal can interfere with the other.

Definition 5.12. Goals g_1 and g_2 are *search independent* for constraint c and program P iff

$$\forall c_1 \in pans_P(\langle g_1, c\rangle) \text{ and } \forall c_r \in pans_P(\langle g_2, c\rangle), consistent(c_1 \wedge c_r).$$

A collection of goals $g_1 : \cdots : g_n$ is search independent for a given c and P iff for every $g_i, 1 \leq i \leq n$: g_i and any goal formed with goals from $g_1 : \cdots : g_{i-1} : g_{i+1} : \cdots : g_n$ are search independent for c and P. Also, a collection of goals is search independent for a set of constraints (interpreted as their disjunction) C and P iff they are search independent for each $c \in C$ and P. Finally, a collection of goals is simply search independent for P iff they are search independent for the set of all possible constraints and P.$\triangle$

Then, in the same spirit as Theorem 5.8 we can conclude:

Corolary 5.13. Goals g_1 and g_2 are search independent for constraint c and program P iff

$$\forall c_1 \in ans_P(\langle g_1, c\rangle), \text{the search spaces of } \langle g_2, c\rangle \text{ and } \langle g_2, c_1\rangle \text{ are the same, and}$$

$$\forall c_r \in ans_P(\langle g_2, c\rangle), \text{the search spaces of } \langle g_1, c\rangle \text{ and } \langle g_1, c_r\rangle \text{ are the same. } \triangle$$

6 ENSURING INDEPENDENCE "A PRIORI"

While compile-time detection of independence can be based on the previous definitions themselves, practical run-time detection cannot. This is because independence has been defined in terms of the answers and partial answers produced by the goals, but, in practice, we are interested in an "a priori" detection of independence conditions (i.e., when detection must be performed just before executing the goals, and without actually having to execute them). In order to do this, (run-time) conditions for ensuring independence must be based only on information which is readily available before executing the goals, namely, the current constraint store and the variables appearing in the goals. One consequence is that an a priori test will not be able to distinguish between the various notions of independence—weak, strong, and search—introduced before.

Our first approach is to define conditions which must hold for each constraint defined over the variables of each goal:

Definition 6.1. Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are *projection independent* for constraint c iff for all constraints c_1 and c_2, if $consistent(c \wedge \exists_{-\bar{x}} c_1)$ and $consistent(c \wedge \exists_{-\bar{y}} c_2)$ hold then $consistent(c \wedge \exists_{-\bar{x}} c_1 \wedge \exists_{-\bar{y}} c_2)$ also holds. A collection of goals g_1 :

$\cdots : g_n$ is projection independent for a given c iff for every $g_i, 1 \leq i \leq n$: g_i and the goal $g_1 : \cdots : g_{i-1} : g_{i+1} : \cdots : g_n$ are projection independent for c and P. Also, a collection of goals is projection independent for a set of constraints (interpreted as their disjunction) C iff they are projection independent for each $c \in C$. Finally, a collection of goals is simply *projection independent* iff they are projection independent for the set of all possible constraints. $\triangle$

Since the execution of a goal can only add constraints on local variables and the arguments of the goal, it is straightforward to prove the following result:

THEOREM 6.2. *Goals g_1 and g_2 are search independent for constraint c and any program P if they are projection independent for c.* $\triangle$

It follows that, since search independence implies strong independence, which in turn implies weak independence, projection independence also implies weak and strong independence.

Naive application of the definition of projection independence implies testing all possible consistent constraints over the variables of each goal. A more useful characterization of projection independence follows. It is based on identifying those variables which are "fixed" in a constraint c where a variable x is *fixed* in c if c implies that x has a single value. We let $\mathit{fixed}(c)$ denote the set of fixed variables in c.

THEOREM 6.3. *Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for constraint c if*

$$(\bar{x} \cap \bar{y} \subseteq \mathit{fixed}(c)) \text{ and } (\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c \rightarrow \exists_{-\bar{y} \cup \bar{x}} c).^6$$

PROOF. Assume that the condition holds but there exist two constraints c_1 and c_2 such that both $c \wedge \exists_{-\bar{x}} c_1$ and $c \wedge \exists_{-\bar{y}} c_2$ are consistent but $c \wedge \exists_{-\bar{x}} c_1 \wedge \exists_{-\bar{y}} c_2$ is not. By assumption $\bar{x} \cap \bar{y} \subseteq \mathit{fixed}(c)$, and therefore $\exists_{-\bar{x}} c_1 \wedge \exists_{-\bar{y}} c_2$ must be consistent. Also by assumption, $\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c$ implies $\exists_{-\bar{y} \cup \bar{x}} c$, and therefore $\exists_{-\bar{y} \cup \bar{x}} c \wedge \exists_{-\bar{x}} c_1 \wedge \exists_{-\bar{y}} c_2$ is consistent, which contradicts the assumption that $c \wedge \exists_{-\bar{x}} c_1 \wedge \exists_{-\bar{y}} c_2$ is inconsistent. $\square$

This condition is not only sufficient but also necessary for projection independence whenever the constraint domain is sufficiently expressive, that is,

Definition 6.4. A constraint domain has *nameable elements* if

- for any constraint c and variable x, if x is not fixed in c, then there exist primitive constraints of form $x = l_1$ and $x = l_2$ where l_1 and l_2 are variable-free expressions such that $c \wedge x = l_1$ and $c \wedge x = l_2$ are consistent but $c \wedge x = l_1 \wedge x = l_2$ is not consistent; and
- if $c_1 \not\leftrightarrow c_2$, where c_1 and c_2 are conjunctions of possibly existentially quantified primitive constraints, then there exists a conjunction of primitive constraints, d, of form $x_1 = l_1 \wedge \cdots \wedge x_2 = l_2$ where the x_i are distinct variables and the l_i are variable-free expressions such that $c_1 \wedge d$ is satisfiable but $c_2 \wedge d$ is not. $\triangle$

These conditions are satisfied by any constraint domain which has an expression (i.e., a "name") for each value in its domain which can be distinguished by the

[6] Note that $(\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c \leftarrow \exists_{-\bar{y} \cup \bar{x}} c)$ always holds. $\triangle$

primitive constraints. To the best of our knowledge, all constraint domains used in practice have nameable elements. As an example of a constraint domain without nameable elements, consider a restricted integer constraint domain where the only primitive constraint is equality and the only non variable expression is 0. Then a variable might not be fixed, but we still cannot find two values which satisfy the first condition above.

THEOREM 6.5. *For constraint domains with nameable elements, goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for constraint c only if*

$$(\bar{x} \cap \bar{y} \subseteq \mathit{fixed}(c)) \text{ and } (\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c \rightarrow \exists_{-\bar{y} \cup \bar{x}} c).\triangle$$

PROOF. Let us reason by contradiction and assume that although $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for c, there exists $z \in \bar{x} \cap \bar{y}$ such that $z \notin \mathit{fixed}(c)$. Then, from the nameable element assumption there must exist variable free expressions l_1 and l_2 such that $c \wedge (z = l_1)$ and $c \wedge (z = l_2)$ are consistent but $c \wedge (z = l_1) \wedge (z = l_2)$ is not consistent. Therefore, $c_1 \equiv (z = l_1)$ and $c_2 \equiv (z = l_2)$ do not satisfy the conditions required by projection independence, and there is a contradiction.

Now assume that $(\bar{x} \cap \bar{y} \subseteq \mathit{fixed}(c))$ but $(\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c) \not\rightarrow \exists_{-\bar{y} \cup \bar{x}} c$ then, from the nameability assumption there must exist two sequences of variable free expressions $\bar{l_x}$ and $\bar{l_y}$ such that (1) $(\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c) \wedge \bar{x} = \bar{l_x} \wedge \bar{y} = \bar{l_y}$ is consistent but (2) $(\exists_{-\bar{y} \cup \bar{x}} c) \wedge \bar{x} = \bar{l_x} \wedge \bar{y} = \bar{l_y}$ is not consistent. It follows from (1) that both $c \wedge \bar{x} = \bar{l_x}$ and $c \wedge \bar{y} = \bar{l_y}$ are consistent. Together with (2) this means that, $c_1 \equiv (\bar{x'} = \bar{l_x})$ and $c_2 \equiv (\bar{y'} = \bar{l_y})$ do not satisfy the conditions required by projection independence, and there is a contradiction. □

Intuitively, the above proof is based on the fact that the only way in which variables in $\bar{x}$ can affect the values of the variables $\bar{y}$ (and vice-versa), is by either (a) having non fixed variables in common or (b) appearing together in some "relevant" constraint. The condition above specifically eliminates these two possibilities.

Example 6.6. Consider the goals $g_1(x, y), g_2(z, w)$ and the constraint $x + y + z + w = 7$. It is obvious that the goals satisfy (a) but not (b) above, since the constraint $x + y + z + w = 7$ is relevant for the relationship between the variables in $\{x, y\}$ and the variables in $\{z, w\}$. But if we add the constraint $x + y = 5$, then the old constraint $x + y + z + w = 7$ becomes irrelevant (it can be substituted by $z + w = 2$) since there is no longer a relationship between the variables in $\{x, y\}$ and the variables in $\{z, w\}$.△

Corolary 6.7. Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are search independent for constraint c and any program P if $\bar{x} \cap \bar{y} \subseteq \mathit{fixed}(c)$ and $\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c \rightarrow \exists_{-\bar{y} \cup \bar{x}} c$ △

The proof comes directly from Theorems 6.2 and 6.3.

Example 6.8. Consider the goals $g_1(y)$ and $g_2(z)$ and constraint $c \equiv y > x, z > x$. Now $\exists_{-\{y\}} c = \epsilon$, $\exists_{-\{z\}} c = \epsilon$, $\exists_{-\{y,z\}} c = \epsilon$. Therefore, from Corollary 6.7, we know that $g_1(y)$ and $g_2(z)$ are search independent for c.△

The following theorem provides the link with the sufficient conditions defined in Hermenegildo and Rossi [1995] (briefly summarized in Section 2.2) for logic programs.

THEOREM 6.9. *In the context of term equations, two goals are projection independent for constraint c iff they are strictly independent for mgu(c).* $\triangle$

PROOF. There is a natural bijection between solved form equations and (idempotent) substitutions. If θ is $\{x_1 \mapsto t_1, ..., x_n \mapsto t_n\}$ we define $cons(\theta)$ to be the solved form equation $x_1 = t_1 \wedge \cdots \wedge x_n = t_n$.

It follows from the definition of strict independence that two goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are strictly independent for c iff the sets

$$(\{x \in \bar{x} |\ \nexists (x = t) \in c'\} \cup \bigcup_{x \in \bar{x}} \{vars(t) | (x = t) \in c'\})$$

and

$$(\{y \in \bar{y} |\ \nexists (y = t) \in c'\} \cup \bigcup_{y \in \bar{y}} \{vars(t) | (y = t) \in c'\})$$

are disjoint, where c' is $cons(mgu(c))$. Straightforward, but tedious, case analysis shows that the sets are disjoint holds iff c' is projection independent. Since c is equivalent to c', it follows that the sets are disjoint iff c is projection independent. $\square$

Thus projection independence is the natural generalization of strict independence to arbitrary constraint domains. As a consequence of the above theorem, for term equations there is no need to actually project the constraint store over any set of variables. However, when constraint domains other than Herbrand are involved, the cost of performing a precise projection may be too high. For example, projection over the linear arithmetic constraints has exponential complexity.

A pragmatic solution is to find if variables are "linked" through the primitive constraints in the constraint store. In fact we can do better by noticing that we can ignore variables that are constrained to take a unique value. Note that in the following we will treat a constraint c as a syntactic object, rather than in terms of its logical meaning.

More formally, the relation $link_c(x, y)$ holds for variables x and y if there is a primitive constraint c' in c such that $\{x, y\} \subseteq vars(c') \setminus fixed(c)$. The relation $links_c(x, y)$ is the transitive closure of $link_c(x, y)$. We lift $links$ to sets of variables by defining $Links_c(\bar{x}, \bar{y})$ iff $\exists x \in \bar{x}$ and $\exists y \in \bar{y}$ such that $links_c(x, y)$.

THEOREM 6.10. *Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for constraint c if $\neg Links_c(\bar{x}, \bar{y})$.* $\triangle$

Note that the above theorem does not depend on the syntactic representation we choose for c. In fact, if the solver keeps a "normal form" for the current constraints, we are better off using the normal form rather than the original sequence of constraints as this allows the definition to be simplified. More precisely, constraints c are in *normal form* if they have form

$$x_1 = f_1(\bar{y}) \wedge x_2 = f_2(\bar{y}) \wedge ... \wedge x_n = f_n(\bar{y}) \wedge c'$$

where the f_i are function symbols of the underlying constraint domain, the x_i are distinct variables disjoint from the variables $\bar{y}$ and $vars(c') \subseteq \bar{y}$. Associated with the normal form is an assignment ψ to the eliminated variables, namely,

$\{x_1 \mapsto f_1(\bar{y}), ..., x_n \mapsto f_n(\bar{y})\}$. It is straightforward to verify that $Links_c(\bar{x}, \bar{y})$ iff $Links_{c'}(vars(\psi(\bar{x})), vars(\psi(\bar{y})))$.

The condition imposed by Theorem 6.10, although clearly sufficient, is somewhat conservative. For instance, although the goals $g_1(y)$ and $g_2(z)$ are search independent for $c \equiv y > x \wedge z > x$, $Links_c(\{y\}, \{z\})$ holds due to the transitive closure performed when computing $links_c(y, z)$. Thus, if projection may be efficiently performed for the particular constraint domain and solver, it is better to use the conditions in Theorem 6.3 to determine search independence at run time.

One important issue that remains to be discussed is the practical usefulness of the different independence notions. In the context of traditional logic programs, it has been shown [Bueno et al. 1999] that strict independence can be detected at compile-time with reasonable accuracy and can be proved at run-time without introducing great overheads. Non a priori notions are, however, a different issue since they cannot be used as the basis of run-time test and, thus, must be detected at compile-time. Unfortunately, even restricted non a priori notions, such as non strict independence, are difficult to detect accurately at compile-time [Cabeza and Hermenegildo 1994]. This is not only because they require more complex analysis domains, but also because they require a particular analysis framework in which literals in the body of a clause are analyzed both in the context of the answers of previous literals and in isolation. In order not to exponentially increase the complexity, a possible approach would be to first perform a typical analysis, use that information to discard literals over which the desired optimization cannot be performed, and then re-analyze the rest in isolation.

In the case of other CLP languages, the problems found for non a priori independence notions become even more acute, due to the complexity of the domains needed to detect consistency of constraints accurately. In the case of a priori notions, preliminary experiments in García de la Banda et al. [1996] show that traditional groundness and freeness information can be used to detect a priori independence at compile-time, but for accuracy, analysis domains specialised to the particular constraint domain used by the program, should be used. Regarding the comparison between the run-time test based on projection and link independence, the experimental evaluation showed two interesting conclusions. First, although there exist cases in which projection independence detects parallelism which link independence fails to detect, this is not a common case. Second, naive implementations of the independence tests introduce too much overhead, especially for projection independence.

7 SOLVER INDEPENDENCE

From the results in previous sections, it may be thought that search space independence is enough for ensuring not only the correctness but also the efficiency of any transformation applied to the search independent goals. Unfortunately, as mentioned in Section 5, this is not true in general.

Example 7.1. Consider the state $\langle \texttt{p(x)} : \texttt{q(x,y,z,w)}, true \rangle$ and the program P:

```
p(x)          ← x = 1.
q(x,y,z,w)    ← x + 1 = y, y = 2 + z, 2 * x + y = w, y > x, w > z.
```

It is easy to see that `p(x)` and `q(x,y,z,w)` are search space independent for *true* and P. However, executing $\langle \mathtt{q(x,y,z,w)}, \mathit{true} \rangle$ is more expensive than executing $\langle \mathtt{q(x,y,z,w)}, \mathtt{x} = \mathtt{1} \rangle$. The reason is that while in the former case a relatively complex constraint solving algorithm (such as the *Simplex* introduced later) has to be applied, in the latter only simple value propagation is needed. As a result, the parallelization or the reordering of the goals in the above state may actually produce a slowdown.$\triangle$

The problem is that the amount of work performed when applying a particular transition rule is not always independent of the state to which this transition rule is applied. There are two different cases. On one hand, given a program P and a state s in which the leftmost literal is an atom a, the amount of work performed when applying the $\rightarrow_r$ or $\rightarrow_{rf}$ transition rules to s is identical to that performed when applying the same transition rule to state s' as long as the leftmost literal in the sequence of literals of s' is a variant of a, since the constraint stores in s and s' are not taken into account. On the other hand, given a program P and a state s in which the leftmost literal is a constraint c', the amount of work performed when applying the $\rightarrow_c$ and $\rightarrow_{cf}$ transition rules to s can be different to that performed when applying the same transition rule to state s' even if the leftmost literal in the sequence of literals of s' is a variant of c'. The key is in the differences between the constraint stores of s and s'.

Therefore, although as shown in the previous sections search space preservation ensures that for each transition rule the number of applications of this transition rule in the derivation trees of each state is preserved, it does not ensure the preservation of the amount of work when the $\rightarrow_c$ and $\rightarrow_{cf}$ transition rules are applied and the store in each state is different. The main problem is that the constraint solver is viewed as a black box, i.e., the operational semantics allow us to see the transitions applied at the higher level, but not those performed by the constraint solver at each of those high level transitions. If we could have access to such "low level" transitions, the amount of work performed by the constraint solver in adding a particular constraint to a particular store, would become explicit, and it could be characterized in terms of search space, analogously as for the high level transitions. This is in fact the approach taken in Bueno et al. [1998].

Therefore, it is clear 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. In fact, this issue is the core of the reordering transformation described in Marriott and Stuckey [1992]. This variance in the cost of adding primitive constraints to the store has been ignored as a factor of negligible influence in traditional logic programming. This is due to the specific characteristics of the standard unification algorithms [Paterson and Wegman 1978; Martelli and Montanari 1982] – we will return to this point later. However, as shown before, it cannot be ignored in the context of other CLP languages. For this reason, we now introduce the notion of constraint solver independence, a new type of independence which, although orthogonal to search space independence, is also needed in order to ensure the efficiency of transformations such as goal reordering and independent and-parallelization.

Intuitively, two sequences of primitive constraints are independent of each other for a given solver if adding them to the current constraint store in any "merging" has the same overall cost. We now make this idea more precise. Let *Solv* be a particular constraint solver and c and c' sequences of primitive constraints. We let $scost(Solv, c, c')$ be the cost of adding the sequence c' to the solver *Solv* after c has been added. To illustrate the vagaries of constraint solving we note that even in "reasonable" constraint solvers such as, for example, that employed in CLP($\Re$), we do *not* have that, if c'' is a subsequence of c',

$$scost(Solv, c, c'') \leq scost(Solv, c, c'),$$

as witness Example 7.1 above.

We let $merge(c, c')$ be the set of all mergings of the constraint sequences c and c'.

Definition 7.2. Constraint sequences c' and c'' are *K-independent* for store c and solver *Solv* iff $consistent(c' \wedge c'' \wedge c)$ implies that for every $c_1, c_2 \in merge(c', c'')$, $scost(Solv, c, c_1) - scost(Solv, c, c_2) \leq K$. $\triangle$

The intuition behind the parameterization of the definition is that the cost be bound by, for instance, a constant value or perhaps a linear function of the number of shared variables among the sequences, where different levels of cost can be tolerated by different applications, also depending on the constraint system being used.

The obvious way to define independence for a solver is by ensuring that adding any pair of consistent sequences of constraints in any order leads to only small differences in cost. This is captured in the following definition.

Definition 7.3. A constraint solver *Solv* is *independent* iff for all constraint sequences c, c' and c'', c' and c'' are K-independent for c and *Solv*, where K is a "small" constant value.$\triangle$

Unfortunately, many reasonable constraint solvers do not satisfy solver independence. In many applications a weaker notion is acceptable, namely that the solver should be solver independent only for sequences which do not "interfere." This notion is inspired by the kind of constraints obtained from projection independent goals (or their almost equivalent characterization provided by Theorems 6.3 and 6.5).

Definition 7.4. A constraint solver *Solv* is *projection independent* iff for all constraint sequences c, c', and c'', if $vars(c') \cap vars(c'') \subseteq fixed(c)$ and $\exists_{-vars(c')} c \wedge \exists_{-vars(c'')} c \rightarrow \exists_{-vars(c') \cup vars(c'')} c$, then c' and c'' are K-independent for c and *Solv*, where K is a "small" constant value.$\triangle$

An even weaker notion of independence holds if we only consider constraints whose variables are not linked in any way through the store.

Definition 7.5. A constraint solver *Solv* is *link independent* iff for all constraint sequences c, c' and c'', if $\neg Links_c(vars(c'), vars(c''))$ then c' and c'' are solver K-independent for c and *Solv*, where K is a "small" constant value.$\triangle$

In practice link independence seems more useful than projection independence: we claim that most reasonable constraint solvers are link independent and that, therefore, the efficiency of many optimizations, such as and-parallelism, can be ensured once the adequate a priori notion is proved to hold for the goals involved in the optimization.

In order to exemplify the applicability of the previously defined notions we will review a few examples of solvers with respect to their solver independence characteristics.

In many CLP systems, for example CLP($\Re$) and Prolog-III [Colmerauer 1990], constraint testing over systems of linear equations and inequations is performed using an incremental version of the simplex algorithm [Marriot and Stuckey 1998]. Essentially this involves incrementally recomputing a normal form for the constraint store when a new constraint is added. This is done by a succession of "pivots" which exchange the variables being eliminated. When a constraint is first encountered it is "simplified" by eliminating the variables from it. If this reduces the constraint to a simple assignment or Boolean test, then, for efficiency, the constraint is not passed to the constraint solver but is handled by the constraint solver "interface." In order to recognize such assignments or tests the solver keeps track of all variables which are constrained to a fixed value. For efficiency, the normal form "tableaux" is stored using a sparse matrix representation as a list of lists of non zero entries. Let this constraint solver be called *Simplex*.

It is easy to construct examples showing that *Simplex* is neither independent nor projection independent. However, we do have that *Simplex* is link independent. This is because if the constraints in c' and c'' are not linked through c, then their normal form in the tableaux does not share variables. Since the tableau is stored using a sparse representation, pivoting or eliminating a variable in an equation derived from c' will not consider equations derived from c'', since they do not share non zero entries in the tableau (and vice versa). If c' and c'' share variables that have a fixed value then these variables will essentially be eliminated from the tableau and replaced by their value thus removing the connection between them.

We believe that the reason *Simplex* is link independent is typical of many solvers used in practice. It is instructive to reconsider unification algorithms as solvers for equality constraints over the domain of Herbrand terms and study their independence characteristics. It is clear that most reasonable unification algorithms would satisfy the conditions of projection independence, and in particular those which are "linear," i.e., which have the property of performing a number of atomic steps which is linear in the size of the terms being unified [Paterson and Wegman 1978; Martelli and Montanari 1982]. Furthermore, if we denote by $LinUnif$ a unification algorithm belonging to the latter class, then we have that $LinUnif$ is independent.

It is interesting to point out that independence does not hold even for all term equation solvers. For example, the cost of the original unification algorithm of Robinson [1965], which is exponential in the worst case, can vary by more than a constant factor depending on reordering. The algorithm used in most practical logic programming systems is actually an adaptation of Robinson's. However, these algorithms can actually be linear because either they (incorrectly) do not perform the occur check, and because they do not materialize the substitutions, but rather keep them in an implicit representation using pointers). In fact, in most practical implementations the difference of execution time after reordering will actually be very close to zero. This is the assumption that is used in practice in optimizations of logic programs based on independence, and it is this assumption which makes the classical view of expressing independence in logic programs only in terms of search independence correct.

8 ALLOWING DYNAMIC SCHEDULING

In previous sections we have studied independence for languages in which calls are evaluated using a fixed left-to-right scheduling strategy. Thus, our results do not directly apply to many modern CLP languages, since these often provide a form of co-routining called *dynamic scheduling*. That is, in these languages the default scheduling is still left-to-right, but some calls may be dynamically "delayed" until their arguments are sufficiently instantiated.

We now extend our independence results to dynamically scheduled languages. This is important for at least four reasons. First, as we have suggested many existing CLP languages already provide flexible scheduling. Therefore, in order to be practical, independence-based optimizations must handle dynamically scheduled programs. Second, dynamic scheduling has a significant cost, increasing the need of applications which improve efficiency. Third, dynamically scheduled languages are considered as promising target languages for the implementation of concurrent constraint logic languages [Debray 1993; Saraswat 1987; Ueda and Chikiyama 1985]. Fourth, dynamic scheduling is also present in some logic programming languages but has been ignored in work on parallelization of logic programs.

8.1 Operational Semantics

The operational semantics of CLP programs with dynamic scheduling is slightly more complex than that given earlier for CLP languages with fixed left-to-right literal scheduling. Again the semantics may be presented as a transition system on *states*. However, a non fail state $\langle G, c, D\rangle$ now also contains a sequence of delayed atoms, D. The other components—the sequence of non executed literals, G, and the constraint store, c—remain the same. In addition to the constraint solving function $consistent(c)$, the operational semantics is parameterized by the predicate $delay(a, c)$, which holds iff a call to atom a delays with the constraint c, and the function $woken(D, c)$ which returns the sequence of atoms in D which are woken for constraint c. Note that the order of the atoms returned by $woken$ is system dependent. Furthermore, the parametric functions $delay$ and $woken$ are assumed to satisfy the following five conditions:

- If $G = woken(D, c)$, then $a \in G$ iff $a \in D \wedge \neg delay(a, c)$.
- If $G_1 = woken(D_1, c)$ and $G_2 = woken(D_2, c)$, D_1 is a subsequence of D_2 and for all $a \in D_1 \setminus D_2$, $delay(a, c)$ holds, then $G_1 \equiv G_2$.
- Let ρ be a renaming, then $delay(a, c)$ iff $delay(\rho(a), \rho(c))$.
- $delay(a, c)$ iff $delay(a, \exists_{-vars(a)} c)$.
- If $c \rightarrow c'$ and $delay(a, c)$, then $delay(a, c')$.

The first condition ensures that there is a congruence between the conditions for delaying an atom and waking it. The second condition ensures that the order of woken goals only depends on their relative ordering in the sequence of delayed goals. The remaining conditions ensure that $delay$ behaves reasonably, i.e., it does not take variable names into account, is only concerned with the effect of c on the variables in a, and if an atom is not delayed, adding more constraints never causes it to delay

Let a denote an atom and c' a constraint. The transition rules are

- $\langle a : G, c, D\rangle \rightarrow_d \langle G, c, a : D\rangle$ if $delay(a, c)$ holds;
- $\langle a : G, c, D\rangle \rightarrow_r \langle B :: G, c, D\rangle$ if $delay(a, c)$ does not hold and $B \in defn_P(a)$;
- $\langle a : G, c, D\rangle \rightarrow_{rf} fail$ if $delay(a, c)$ does not hold and $defn_P(a) = \emptyset$;
- $\langle G, c, D\rangle \rightarrow_w \langle G' :: G, c, D \setminus G'\rangle$ where $G' = woken(D, c)$, and G' is not empty (note that $\setminus$ is assumed to preserve the relative order of the unwoken goals);
- $\langle c' : G, c, D\rangle \rightarrow_c \langle G, c \wedge c', D\rangle$ if $consistent(c \wedge c')$ holds;
- $\langle c' : G, c, D\rangle \rightarrow_{cf} fail$ if $consistent(c \wedge c')$ does not hold.

The above operational semantics extends that given earlier for traditional CLP languages by adding the $\rightarrow_d$ and $\rightarrow_w$ transitions. Note that the conditions for applying each of the transition rules are still pairwise exclusive, except for $\rightarrow_w$. We will assume that this transition rule has preference over the rest of the rules, thus being applied to any state $\langle G, c, D\rangle$ in which $woken(D, c)$ is non empty. The same result could also be achieved by combining the $\rightarrow_c$ and $\rightarrow_w$ transition rules, but given our interest in the particular characteristics of $\rightarrow_w$, we have kept them separate. As before, we will assume a depth-first search strategy. All notions related to derivations and derivation trees are straightforward extensions of those defined for the CLP context. One difference is that answers may now contain a sequence of delayed goals and so consist of tuples whose first element is the answer constraint and whose second element is the delayed goal sequence.

We now generalize our earlier results to CLP languages with dynamic scheduling. We do this in two stages. First we assume that the initial sequence of delayed goals is empty. In the second stage we shall drop this assumption.

8.2 Independence When the Initial Sequence D is Empty

We start by extending our and-parallel execution model to this new context. Assume that given the program P and the state $\langle g_1 : g_2 : G, c, nil\rangle$, we want to execute g_1 and g_2 in parallel. Then the execution scheme is the following:

- execute $\langle g_1, c, nil\rangle$ and $\langle g_2, c, nil\rangle$ in parallel (in different environments) obtaining the answers $\langle c_1, D_1\rangle$ and $\langle c_r, D_r\rangle$ respectively,
- obtain $c_s = c_1 \wedge c_r$, and
- execute $\langle G, c_s, D_r :: D_1\rangle$.

Modulo the extra transformation rules, the definition of search space preservation remain the same as that given earlier. The definition of correctness is extended as follows:

Definition 8.1. Let s be the state $\langle g_1 : g_2 : G, c, nil\rangle$ and P a program. The parallel execution of g_1 and g_2 is *correct* iff for every $\langle c_1, D_1\rangle \in ans_P(\langle g_1, c, nil\rangle)$ there exists a renaming γ such that $\gamma(s) \equiv s$ and a bijection which assigns to each answer $\langle c_s, D_s\rangle \in ans_P(\langle g_2, c_1, D_1\rangle)$, an answer $\langle c_t, D_t\rangle \in ans_P(\langle nil, c_1 \wedge c_r, D_r :: D_1\rangle)$ such that $D_s \equiv \gamma(D_t)$ and $c_s \equiv \gamma(c_t)$, where $\langle c_r, D_r\rangle \in ans_P(\langle g_2, c, nil\rangle)$.$\triangle$

Note that the above definition allows goals in $D_r :: D_1$ to be woken up due to the conjunction of c_1 and c_r performed once the parallel execution is finished. We similarly extend the definition of operational correctness and the definition of efficiency as follows

Definition 8.2. Let s be the state $\langle g_1 : g_2 : G, c, nil\rangle$ and P be a program. The parallel execution of g_1 and g_2 is *efficient* iff for every $\langle c_1, D_1\rangle \in ans_P(\langle g_1, c, nil\rangle)$:

$$cost(\langle g_2, c, nil\rangle) + \sum_{\langle c_r, D_r\rangle \in ans_P(\langle g_2, c, nil\rangle)} cost(\langle nil, c_1 \wedge c_r, D_r :: D_1\rangle) \leq cost(\langle g_2, c_1, D_1\rangle)$$

$\triangle$

It is clear from the definition of correctness and efficiency, that one of the main differences between the frameworks obtained for languages with and without dynamically scheduled is that in the former case the state obtained by conjoining the answers obtained by the parallel execution ($\langle nil, c_1 \wedge c_r, Dr :: D_1\rangle$) might not be a final state due to the awakening of some previously delayed goal. It is also clear that this difference is going to make things complicated, since we not only have to directly compare the sequential execution with the parallel but also have to take into account the execution of the atoms woken due to the conjoining of the parallel answers. It is thus tempting to believe that, by requiring the state $\langle nil, c_1 \wedge c_r, Dr :: D_1\rangle$ to be final, we return to a situation to which the results of the previous sections can be easily extended. Certainly this is the case for proving that search space preservation is sufficient for ensuring efficiency:

THEOREM 8.3. *Let P be a program and $\langle g_1 : g_2 : G, c, nil\rangle$ a state. The parallel execution of g_1 and g_2 is efficient if for every $\langle c_1, D_1\rangle \in ans_P(\langle g_1, c, nil\rangle)$, the state $\langle nil, c_1 \wedge c_r, Dr :: D_1\rangle$ is final and the search spaces of $\langle g_2, c, nil\rangle$ and $\langle g_2, c_1, D_1\rangle$ are the same for P.*$\triangle$

The proof is straightforward, since search space preservation ensures the existence of a bijection among transitions of each particular kind. However, even in this restricted context, proving that search space preservation is sufficient for guaranteeing correctness is much more involved. Previously, the reasoning was based on the fact that, in absence of failure, for every two nodes r in $tree_P(\langle g_2, c\rangle)$ and s in $tree_P(\langle g_2, c_1\rangle)$ with the same path, their sequence of selected literals must be identical up to renaming. Thus, the constraints added to the store are also the same, up to renaming. Unfortunately, we can no longer guarantee that the atoms are woken in the same order in both executions. Thus the sequence of active literals in nodes with the same path can differ.

Example 8.4. Consider the parallel execution of the goals in state $\langle \mathtt{p(x,y)} : \mathtt{q(x,y)}, \mathtt{true}, \mathtt{nil}\rangle$ for the program P:

```
p(x,y) ← x=y.                          s(x,y,z) ← f(0,0)=f(y,z).
q(x,y) ← r(y,z),s(x,y,z),t(z).         t(z)     ← z=0.
r(y,z) ← y=z.
```

with the following suspension declarations for `p/2, q/2, r/2, s/3` and `t/1`:

$? - \mathtt{r(y,z)}$ `when ground(y).`

$$
\begin{gathered}
\langle \mathtt{q(x,y)}, \mathtt{x = y}, \mathtt{nil} \rangle \\
\downarrow_r \\
\langle \mathtt{r(y,z) : s(x,y,z) : t(z) : x = 0}, \mathtt{x = y}, \mathtt{nil} \rangle \\
\downarrow_d \\
\langle \mathtt{s(x,y,z) : t(z) : x = 0}, \mathtt{x = y}, \mathtt{r(y,z)} \rangle \\
\downarrow_d \\
\langle \mathtt{t(z) : x = 0}, \mathtt{x = y}, \mathtt{r(y,z) : s(x,y,z)} \rangle \\
xe \downarrow_d \\
\langle \mathtt{x = 0}, \mathtt{x = y}, \mathtt{r(y,z) : s(x,y,z) : t(z)} \rangle \\
\downarrow_c \\
\langle \mathtt{nil}, \mathtt{x = y \wedge x = 0}, \mathtt{r(y,z) : s(x,y,z) : t(z)} \rangle \\
\downarrow_w \\
\langle \mathtt{r(y,z) : s(x,y,z)} :, \mathtt{x = y \wedge x = 0}, \mathtt{t(z)} \rangle \\
\downarrow_r \\
\langle \mathtt{y = z : s(x,y,z)}, \mathtt{x = y \wedge x = 0}, \mathtt{t(z)} \rangle \\
\downarrow_c \\
\langle \mathtt{s(x,y,z)}, \mathtt{x = y \wedge x = 0 \wedge y = z}, \mathtt{t(z)} \rangle \\
\downarrow_w \\
\langle \mathtt{t(z) : s(x,y,z)}, \mathtt{x = y \wedge x = 0 \wedge y = z}, \mathtt{nil} \rangle \\
\downarrow_r \\
\langle \mathtt{z = 0 : s(x,y,z)}, \mathtt{x = y \wedge x = 0 \wedge y = z}, \mathtt{nil} \rangle \\
\downarrow_c \\
\langle \mathtt{s(x,y,z)}, \mathtt{x = y \wedge x = 0 \wedge y = z \wedge z = 0}, \mathtt{nil} \rangle \\
\downarrow_r \\
\langle \mathtt{f(0,0) = f(y,z)}, \mathtt{x = y \wedge x = 0 \wedge y = z \wedge z = 0}, \mathtt{nil} \rangle \\
\downarrow_c \\
\langle \mathtt{nil}, \mathtt{x = y \wedge x = 0 \wedge y = z \wedge z = 0 \wedge f(0,0) = f(y,z)}, \mathtt{nil} \rangle
\end{gathered}
$$

Fig. 2.

$? - \mathtt{s(x,y,z)}$ **when** $\mathtt{ground(x)}$.
$? - \mathtt{t(z)}$ **when** $\mathtt{ground(z)}$.

Figure 2 shows the derivation tree from the state $\langle \mathtt{q(x,y)}, \mathtt{x = y}, \mathtt{nil} \rangle$ (i.e., `p(x,y)` has been executed obtaining the answer $\langle \mathtt{x = y}, \mathtt{nil} \rangle$) and Figure 3 the derivation tree from the state $\langle \mathtt{q(x,y)}, \mathtt{true}, \mathtt{nil} \rangle$ (i.e., `p(x,y)` has not been executed). Notice that the search space is identical and that the state resulting from conjoining the answers $\langle \mathtt{nil}, \mathtt{x = y}, \mathtt{nil} \rangle$ and $\langle \mathtt{nil}, \mathtt{x = 0 \wedge f(0,0) = f(y,z) \wedge y = z \wedge z = 0}, \mathtt{nil} \rangle$ of the parallel execution of $\langle \mathtt{p(x,y)}, \mathtt{true}, \mathtt{nil} \rangle$ and $\langle \mathtt{q(x,y)}, \mathtt{true}, \mathtt{nil} \rangle$, respectively, is final.

The first six states in Figure 2 are almost identical to those in Figure 3, the only difference being that the constraint `x = y` already appears in the constraint store of the states in Figure 2. The first important difference appears after the sixth transition due to the fact that, while in Figure 2 variable `y` is known to be ground thus waking up goal `r(y,z)`, in Figure 3 goal `r(y,z)` remains delayed, leading to different sequences of active literals. Even though the search space is preserved, there is no renaming which makes the sequences of active literals identical. Furthermore, there is not even a renaming which makes the leftmost literal of every two non failure nodes with the same path identical. $\triangle$

$$
\begin{array}{c}
\langle \mathtt{q(x,y)}, \mathtt{true}, \mathtt{nil}\rangle \\
\downarrow_r \\
\langle \mathtt{r(y,z)} : \mathtt{s(x,y,z)} : \mathtt{t(z)} : \mathtt{x} = 0, \mathtt{true}, \mathtt{nil}\rangle \\
\downarrow_d \\
\langle \mathtt{s(x,y,z)} : \mathtt{t(z)} : \mathtt{x} = 0, \mathtt{true}, \mathtt{r(y,z)}\rangle \\
\downarrow_d \\
\langle \mathtt{t(z)} : \mathtt{x} = 0, \mathtt{true}, \mathtt{r(y,z)} : \mathtt{s(x,y,z)}\rangle \\
\downarrow_d \\
\langle \mathtt{x} = 0, \mathtt{true}, \mathtt{r(y,z)} : \mathtt{s(x,y,z)} : \mathtt{t(z)}\rangle \\
\downarrow_c \\
\langle \mathtt{nil}, \mathtt{x} = 0, \mathtt{r(y,z)} : \mathtt{s(x,y,z)} : \mathtt{t(z)}\rangle \\
\downarrow_w \\
\langle \mathtt{s(x,y,z)}, \mathtt{x} = 0, \mathtt{r(y,z)} : \mathtt{t(z)}\rangle \\
\downarrow_r \\
\langle \mathtt{f(0,0)} = \mathtt{f(y,z)}, \mathtt{x} = 0, \mathtt{r(y,z)} : \mathtt{t(z)}\rangle \\
\downarrow_c \\
\langle \mathtt{nil}, \mathtt{x} = 0 \wedge \mathtt{f(0,0)} = \mathtt{f(y,z)}, \mathtt{r(y,z)} : \mathtt{t(z)}\rangle \\
\downarrow_w \\
\langle \mathtt{r(y,z)} : \mathtt{t(z)}, \mathtt{x} = 0 \wedge \mathtt{f(0,0)} = \mathtt{f(y,z)}, \mathtt{nil}\rangle \\
\downarrow_r \\
\langle \mathtt{y} = \mathtt{z} : \mathtt{t(z)}, \mathtt{x} = 0 \wedge \mathtt{f(0,0)} = \mathtt{f(y,z)}, \mathtt{nil}\rangle \\
\downarrow_c \\
\langle \mathtt{t(z)}, \mathtt{x} = 0 \wedge \mathtt{f(0,0)} = \mathtt{f(y,z)} \wedge \mathtt{y} = \mathtt{z}, \mathtt{nil}\rangle \\
\downarrow_r \\
\langle \mathtt{z} = 0, \mathtt{x} = 0 \wedge \mathtt{f(0,0)} = \mathtt{f(y,z)} \wedge \mathtt{y} = \mathtt{z}, \mathtt{nil}\rangle \\
\downarrow_c \\
\langle \mathtt{nil}, \mathtt{x} = 0 \wedge \mathtt{f(0,0)} = \mathtt{f(y,z)} \wedge \mathtt{y} = \mathtt{z} \wedge \mathtt{z} = 0, \mathtt{nil}\rangle
\end{array}
$$

Fig. 3.

Even with the problems illustrated in the example above, it is possible to prove that in this restricted context (i.e., the conjunction of the parallel answers does not cause the awakening of an atom delayed during the parallel execution) search space preservation is *sufficient for preserving correctness*. However, we will not do so, since there is a more important problem we must consider: search space preservation no longer ensures operational correctness. 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. In the absence of dynamic scheduling, operational correctness essentially came for free due to the existence of a bijection between answers associated to nodes with the same path, and to the properties of nodes with the same path ensured by Lemma 4.3. With dynamic scheduling, however, those properties do not always hold, due to the possible existence of interleavings between goals which have multiple solutions.

Example 8.5. It is simple to extend the program in Example 8.4 so that it exhibits such behavior (we simply add one extra argument to `r/2` and `s/3` in which we return more than one answer and add an extra rule to each). Consider the parallel execution of the goals in state $\langle \mathtt{p(x,y)} : \mathtt{q(x,y,u,v)}, \mathtt{true}, \mathtt{nil}\rangle$ for the program P:

```
p(x,y)     ← x=y.                          s(x,y,z,v)← f(0,0,1)=f(y,z,v).
q(x,y,u,v)← r(y,z,u),s(x,y,z,v),t(z).  s(x,y,z,v)← f(0,0,2)=f(y,z,v).
r(y,z,u)   ← f(y,u)=f(z,3).               t(z)       ← z=0.
r(y,z,u)   ← f(y,u)=f(z,4).
```

with the following suspension declarations for p/2, q/4, r/3, s/4 and t/1:

$$?-\mathtt{r(y,z,u)\ when\ ground(y)}.$$
$$?-\mathtt{s(x,y,z,v)\ when\ ground(x)}.$$
$$?-\mathtt{t(z)\ when\ ground(z)}.$$

It is easy to check that, although the search space of the two initial states is preserved and no atom delayed during the execution of $\langle \mathtt{p(x,y)}, \mathtt{true}, \mathtt{nil}\rangle$ and $\langle \mathtt{q(x,y,u,v)}, \mathtt{true}, \mathtt{nil}\rangle$ is woken up during the conjunction of the answers, the sequence of answers obtained for the sequential execution is different to that for parallel execution. This is because in the sequential execution r/3 is executed before s/4 returning the four answers (restricted to the variables u and v) in the order $\mathtt{u}=3 \wedge \mathtt{v}=1$, $\mathtt{u}=3 \wedge \mathtt{v}=2$, $\mathtt{u}=4 \wedge \mathtt{v}=1$, and $\mathtt{u}=4 \wedge \mathtt{v}=2$ while in the parallel execution r/3 is executed after s/4 returning the answers in the order $\mathtt{u}=3 \wedge \mathtt{v}=1$, $\mathtt{u}=4 \wedge \mathtt{v}=1$, $\mathtt{u}=3 \wedge \mathtt{v}=2$, and $\mathtt{u}=4 \wedge \mathtt{v}=2$. $\triangle$

We can avoid such problematic interleavings by ensuring that for every answer $\langle c_1, D_1\rangle$ of $\langle g_1, c, nil\rangle$, no atom in D_1 is woken during the execution of $\langle g_2, c_1, D_1\rangle$, and every atom delayed (woken) at some point of the execution of $\langle g_2, c_1, D_1\rangle$ is also delayed (woken) at the same point of the execution of $\langle g_2, c, nil\rangle$. This is achieved by requiring the following conditions for every two nodes s and r of $tree_P(\langle g_2, c_1, D_1\rangle)$ and $tree_P(\langle g_2, c, nil\rangle)$, respectively, with the same path:

Definition 8.6. Two nodes $s \equiv \langle G_s, c_s, D_s\rangle$ and $r \equiv \langle G_r, c_r, D_r\rangle$ are *equivalent with respect to delay* iff:

- for every $a \in D_s \setminus D_r$: $delay(a, c_s)$ holds,
- for every $a \in D_r \setminus D_s$: $delay(a, c_r)$ holds,
- for every $a \in D_s \cap D_r$: $delay(a, c_r)$ iff $delay(a, c_s)$,
- if $G_s \equiv a : G'_s$ and $G_r \equiv a : G'_r$: $delay(a, c_r)$ iff $delay(a, c_s)$.$\triangle$

These conditions ensure that the extra delayed goals in D_r or D_s are not woken up, and that c_s and c_r have identical behavior with respect to delay both for the delayed atoms they share in common and for their leftmost atom if they share it.

Note that this condition is not the most general possible, since it does not allow the interleaving when, for example, one of the goals involved is single solution, or it affects branches other than non failure, finite branches. However, we believe it is a reasonable compromise for at least two reasons. First, if such a condition is satisfied, the situation becomes equivalent to that for languages with a fixed scheduling, allowing us to extend all results obtained in the previous section to this new context. Second, the above condition, although complex, seems amenable to compile-time verification using global analyzers recently developed for dynamically scheduled languages [Puebla et al. 1997].

The following lemma generalizes Lemmas 4.3 and 4.4. Its proof is similar.

Lemma 8.7. Let P be a program and $\langle g_2, c_1, D_1\rangle$, $\langle g_2, c, nil\rangle$ be two states with $c_1 \rightarrow c$. There exists a renaming γ 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 $\gamma(tree_P(\langle g_2, c, nil\rangle))$, respectively, such that for all ancestors s' and r' of s and r respectively, with the same path, s' and r' are equivalent with respect to delay, $G_s \equiv G_r$, $D_s \equiv D_r :: D_1$, and $c_s \leftrightarrow (c_1 \wedge c_r)$;

—$\gamma(\langle g_2, c_1, D_1\rangle) = \langle g_2, c_1, D_1\rangle$ and $\gamma(\langle g_2, c, nil\rangle) = \langle g_2, c, nil\rangle$;

—γ is its own inverse.$\triangle$

Proof. Let us reason by induction. In the base case the only non failure nodes are the root nodes. It is clear that the conditions are satisfied for the identity renaming γ since we have that $D_1 \equiv nil : D_1$ and, since $c_1 \rightarrow c$, we also have that $c_1 \rightarrow c \wedge c_1$. Consider now that there exists a γ' for which the conditions are satisfied for all ancestors s' and r' of non failure nodes s and r, respectively, and let us prove there also exists a γ for which the conditions are satisfied for s and r. Let $r' \equiv \langle G'_r, c'_r, D'_r\rangle$. Then, by assumption, $s' \equiv \langle G'_r, c_1 \wedge c'_r, D'_r :: D_1\rangle$.

Let us first consider the case in which some atom is woken up. By assumption of equivalence with respect to delay we have that $\forall a \in D_1$: $delay(a, c_1 \wedge c'_r)$ holds. Thus, no atom in D_1 will be awoken in s'. Therefore, if some atom is woken up the atom must belong to D'_r. Now, also by assumption of equivalence with respect to delay we have that $\forall a \in D'_r$: $delay(a, c_1 \wedge c'_r)$ iff $delay(a, c'_r)$. Thus, if some $a \in D'_r$ wakes up in r', it will also wake up in s' and vice-versa. Thus we have that if $G_a \equiv woken(D'_r, c'_r)$ then $G_a \equiv woken(D'_r :: D_1, c_1 \wedge c'_r)$. It follows that, for $\gamma' \equiv \gamma$ we have $r \equiv \langle G_a : G'_r, c'_r, D'_r \setminus G_a\rangle$ and $s \equiv \langle G_a : G'_r, c_1 \wedge c'_r, (D'_r \setminus G_a) :: D_1\rangle$, and thus all conditions remain satisfied.

Let us now consider the case in which no atom is woken up. Let $G'_r \equiv c' : G_r$ where c' is a constraint. Then for $\gamma' \equiv \gamma$ we have that $r \equiv \langle G_r, c' \wedge c'_r, D'_r\rangle$ and $s \equiv \langle G_r, c_1 \wedge c' \wedge c'_r, D'_r :: D_1\rangle$, and thus all conditions remain satisfied. Otherwise, $G'_r \equiv a : G_r$ where a is an atom. By assumption of equivalence with respect to delay of r' and s' we have that $delay(a, c'_r)$ iff $delay(a, c_1 \wedge c'_r)$. If $delay(a, c'_r)$ holds, then for $\gamma' \equiv \gamma$ we have that $r \equiv \langle G_r, c'_r, a : D'_r\rangle$ and $s \equiv \langle G_r, c_1 \wedge c'_r, a : D'_r :: D_1\rangle$, and thus all conditions remain satisfied. If $delay(a, c'_r)$ does not hold γ is built as in Lemma 4.3 and $r \equiv \langle B : G_r, c'_r, D'_r\rangle$ and $s \equiv \langle B : G_r, c_1 \wedge c'_r, D'_r :: D_1\rangle$, and thus all conditions remain satisfied. $\square$

Given the above lemma we can now ensure the following:

Theorem 8.8. *Let P be a program, $\langle g_1 : g_2 : G, c, nil\rangle$ a state, and γ a renaming satisfying Lemma 8.7 for states $\langle g_2, c_1, D_1\rangle$ and $\langle g_2, c, nil\rangle$. Parallel execution of g_1 and g_2 is efficient, correct and operationally correct if for every $\langle c_1, D_1\rangle \in ans_P(\langle g_1, c, nil\rangle)$, the search spaces of $\langle g_2, c, nil\rangle$ 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 $tree_P(\langle g_2, c_1, D_1\rangle)$ and $\gamma(tree_P(\langle g_2, c, nil\rangle))$, s and r are equivalent with respect to delay.*$\triangle$

Proof. By definition of search space preservation, there exists a bijection which assigns to every final state $r \equiv \langle G_r, c_r, D_r\rangle$ in $tree_P(\langle g_2, c\rangle)$ a final state $s \equiv \langle G_s, c_s, D_s\rangle$ with the same path in $\gamma(tree_P(\langle g_2, c_1\rangle))$, thus establishing a bijection among the answers. Also, since $c_1 \in ans_P(\langle g_1, c\rangle)$ we can ensure that $c_1 \rightarrow c$. By assumption s and r are equivalent with respect to delay. Thus, by Lemma 8.7, $c_s \leftrightarrow c_1 \wedge c_r$ and $D_s \equiv D_r :: D_1$, and we have proved correctness. Since the bijection is among answers with the same path, we have also proved operational correctness. Since s is a final state, $\langle nil, c_1 \wedge c_r, D_r :: D_1\rangle$ is also a final state and by Theorem 8.3 we have proved efficiency. □

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

Definition 8.9. Goal g_2 is *strongly independent* of goal g_1 for constraint c, the empty sequence of delayed atoms, and program P iff $\forall\langle c_1, D_1\rangle \in ans_P(\langle g_1, c, nil\rangle)$ and $\forall\langle c_r, D_r\rangle \in pans_P(\langle g_2, c, nil\rangle)$:

- $consistent(c_1 \wedge c_r)$ holds,
- $\forall a \in D_r : delay(a, c_r)$ iff $delay(a, c_1 \wedge c_r)$,
- $\forall a \in D_1 : delay(a, c_1 \wedge c_r)$ holds.△

The definition can be extended to a set of goals analogously to Definition 5.6. We can also extend the definition of weak independence and search independence in a similar fashion. The last two conditions in the above definition can be equivalently expressed as $\langle nil, c_r, D_r\rangle$ and $\langle nil, c_1 \wedge c_r, D_r :: D_1\rangle$ are equivalent with respect to delay. Given this definition, it is easy to prove the following results.

THEOREM 8.10. *Goal g_2 is strongly independent of goal g_1 for constraint c, empty sequence of delayed atoms, and program P if $\forall\langle c_1, D_1\rangle \in ans_P(\langle g_1, c, nil\rangle)$, the search spaces of $\langle g_2, c, nil\rangle$ and $\langle g_2, c_1, D_1\rangle$ are the same for P, and there exists a renaming γ such that for every two non failure nodes s and r with the same path in $tree_P(\langle g_2, c_1, D_1\rangle)$ and $\gamma(tree_P(\langle g_2, c, nil\rangle))$, s and r are equivalent with respect to delay.*△

Corolary 8.11. If goal g_2 is strongly independent of goal g_1 for constraint c, the empty sequence of delayed atoms, and program P then the parallel execution of g_1 and g_2 is correct, operationally correct, and efficient for c and P.△

8.3 Independence in the General Case

We now consider the general case in which the initial sequence of delayed atoms D may be non empty. We must first define 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.

We define the conjoin operation as follows: D_s is obtained in the and-parallel model as

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

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 $D \setminus D_r$) and then add the atoms left delayed by $\langle g_2, c, D\rangle$ which do not belong to the initial sequence (represented by $(D_r \setminus D)$).

With this definition, the results obtained in the previous sections regarding the characteristics of both search space preservation and strong independence can be extended to this new context in a straightforward way.

8.4 Ensuring Independence "A Priori"

As mentioned earlier, it is important to find "a priori" conditions which ensure independence. In this section we extend the earlier notion of projection independence to the broader context of languages with dynamic scheduling. Consider two goals g_1 and g_2 which are projection independent for constraint c. If the sequence of delayed atoms D is empty, 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 atoms 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 whether one is executed before or after the other, thus not affecting the atoms left delayed by the other goal. Formally, this is stated as follows

THEOREM 8.12. *Goal g_2 is search independent of goal g_1 for program P, constraint store c, and empty sequence of delayed atoms D if the goals are projection independent for P and c.*$\triangle$

Of course search independence still implies weak and strong independence.

A difference with respect to the previous cases does arise, however, when the initial sequence of delayed atoms D is not empty. In this case the sufficient condition must take into account the constraints established on the variables which appear in the delayed atoms. The reason is that atoms 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 $\bar{y}$.

The solution proposed is to ensure that D can be partitioned into two sequences in such a way that if we associate them to $g_1(\bar{x})$ and $g_2(\bar{y})$ respectively, the two new goals are projection independent for the given c. While the first sequence corresponds to the delayed literals that depend on $g_1(\bar{x})$, the second one corresponds to those that depend on $g_2(\bar{y})$. If there exist delayed atoms which depend on neither $g_1(\bar{x})$ nor $g_2(\bar{y})$, they can be concatenated to either of the two sequences.

Definition 8.13. Goals g_1 and g_2 are *projection independent* for constraint c and sequence of delayed atoms D iff D can be partitioned into two sequences D_1 and D_2 such that the goals $g_1 : D_1$ and $g_2 : D_2$ are projection independent for c. $\triangle$

THEOREM 8.14. *Goal g_2 is search independent of goals g_1 for constraint c and sequence of delayed atoms D if the goals are projection independent for c and D.*$\triangle$

The proof follows directly from Theorem 8.12. Again, search independence also implies weak and strong independence.

9 CONCLUSIONS AND FUTURE WORK

We have shown how a simple extrapolation of the logic programming-based definitions of independence to CLP turns out to be both not general enough in some cases and erroneous in others, and identified the need in CLP for defining concepts of independence both at the search level and at the solver level. Several such concepts have been presented and shown to be relevant to several classes of applications. We have also proposed sufficient conditions for the concepts of independence proposed, which are easier to detect at run-time than the original definitions. Finally, we have extended our results to deal with CLP languages with dynamic scheduling.

Our results also provide insight into the theory of independence for logic programs. The concepts proposed, when restricted to conventional logic programming, are equivalent to the traditional notions. They make explicit hidden assumptions related to properties of the standard unification algorithm and clarify the relationships between independence and search space preservation.

It is our belief that using the concepts of independence presented the range of applications of independence-related optimizations can be even larger in CLP than in logic programming.

One clear topic for future work is to develop analyses for determining independence at compile-time. One step in this direction is the analysis based on the LSign domain defined in Marriott and Stuckey [1994]. In this case the most straightforward approach is to apply the definitions directly – the fact that the definitions are in terms of the run-time answer constraints is not so much of a problem, since the problem of predicting the state of the store after the execution of the goals is probably no more difficult than determining its state before such execution.

Another clear topic for future work is to apply the results to practical optimization tools for CLP languages. First, we are in the process of developing automatic parallelization tools based on these ideas. It appears feasible to extend the formal techniques that have been developed for this purpose in the context of logic programming, by using the herein proposed notions of independence [Bueno et al. 1999]. And, although the topic certainly requires more study, preliminary experiments confirm that useful speedups can be obtained automatically in practice by parallelizing independent goals [García de la Banda et al. 1996]. Also, it appears possible to exploit more fine-grained forms of and-parallelism, provided the definition of independence is applied at the appropriate level: for example, stream and-parallelism can be exploited in CLP programs by considering the independence notions at the level of individual constraints rather than goals [Bueno et al. 1998].

Second, we have developed tools for reordering-based program optimization in the context of CLP languages without dynamic scheduling [Kelly et al. 1996]. However, reordering is even more interesting for the case of dynamic scheduling because the process of delaying a goal involves run-time overhead. Actually, dynamic scheduling can be seen as a run-time form of goal reordering, while the reordering optimization moves the position of goals at compile-time. The topic of reordering in the context of dynamic scheduling, in which independence can also be instrumental, is dealt with in more detail in Puebla et al. [1997].

Finally, our results can be used to detect "stability" [Janson and Haridi 1991]. This notion is used in the Andorra family of languages in general, and in the AKL language in particular, as the rule for control of one of the basic operations of the language – global forking. This operation amounts to starting and-parallel execution of a goal which is non deterministic. Stability for a goal is defined informally as being in a state in which other goals running in parallel with it will not affect its execution. This is of course an undecidable notion, and in practice sufficient conditions are used in actual implementations. In particular, in the first implementation of AKL, restricted to the Herbrand domain, the stability condition used is actually the classical notion of strict independence for logic programming [Franzen 1992]. Since the AKL language is defined to be a constraint language, the notion of stability has to be generalized to the constraint level. As we have shown, generalization cannot be done by directly applying naive liftings of the logic programming concepts of independence. We believe that the results presented in this paper will be of direct application.

10 ACKNOWLEDGMENTS

We thank Francisco Bueno, Lee Naish, and anonymous reviewers of previous versions of this paper for their comments.

REFERENCES

Apt, K. 1990. Introduction to Logic Programming. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B: Formal Model and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 495–574.

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

Bacon, D., Graham, S., and Sharp, O. 1994. Compiler Transformations for High-Performance Computing. *Computing Surveys 26,* 4 (December), 345–420.

Best, E. and Lengauer, C. 1990. Semantic Independence. *Science of Computer Programming 13*, 23–50.

Bruynooghe, M. and Pereira, L. 1984. Deduction Revision by Intelligent Backtracking. In *Implementations of Prolog*, J. Campbell, Ed. Elliss Horwood, 194–205.

Bueno, F., García de la Banda, M., Hermenegildo, M., and Muthukumar, K. 1999. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *J. Logic Program. 38,* 2 (February), 165–218.

Bueno, F., Hermenegildo, M., Montanari, U., and Rossi, F. 1998. Partial Order and Contextual Net Semantics for Atomic and Locally Atomic CC Programs. *Science of Computer Programming 30*, 51–82. Special CCP95 Workshop issue.

Cabeza, D. and Hermenegildo, M. 1994. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium.* Number 864 in LNCS. Springer-Verlag, Namur, Belgium, 297–313.

Chassin, J. and Codognet, P. 1994. Parallel Logic Programming Systems. *Computing Surveys 26,* 3 (September), 295–336.

Colmerauer, A. 1990. An Introduction to Prolog III. *Commun. ACM 28,* 4, 412–418.

Conery, J. S. 1983. The and/or process model for parallel interpretation of logic programs. Ph.D. thesis, The University of California At Irvine. Tech. Rep. 204.

Conery, J. S. 1987. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors. In *Symposium on Logic Programming* 457–467.

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

DeGroot, D. 1984. Restricted AND-Parallelism. In *International Conference on 5th Generation Computer Systems*. Tokyo, 471–478.

Franzen, T. 1992. Logical Aspects of the Andorra Kernel Language. Draft/Personal communication.

García de la Banda, M., Bueno, F., and Hermenegildo, M. 1996. Towards Independent And-Parallelism in CLP. In *Programming Languages: Implementation, Logics, and Programs.* Number 1140 in LNCS. Springer-Verlag, Aachen, Germany, 77–91.

HARIDI, S. AND JANSON, S. 1990. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the 7th International Conference on Logic Programming.* MIT Press, 31–46.

HERMENEGILDO, M. 1997. Automatic Parallelization of Irregular and Pointer-Based Computations: Perspectives from Logic and Constraint Programming. In *Proceedings of EUROPAR'97.* LNCS, vol. 1300. Springer-Verlag, 31–46. (invited).

HERMENEGILDO, M. AND GREENE, K. 1990. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming.* MIT Press, 253–268.

HERMENEGILDO, M. AND ROSSI, F. 1995. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *J. Logic Program. 22,* 1, 1–45.

JAFFAR, J. AND LASSEZ, J.-L. 1987. Constraint Logic Programming. In *ACM Symp. Principles of Programming Languages.* ACM, 111–119.

JAFFAR, J. AND MAHER, M. 1994. Constraint Logic Programming: A Survey. *J. Logic Program. 19/20*, 503–581.

JAFFAR, J. AND MICHAYLOV, S. 1987. Methodology and Implementation of a CLP System. In *4th International Conference on Logic Programming.* University of Melbourne, MIT Press, 196–219.

JANSON, S. AND HARIDI, S. 1991. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium.* MIT Press, 167–183.

KALÉ, L. V. 1987. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *4th IEEE Symposium on Logic Programming.* IEEE, 125–133.

KELLY, A., MACDONALD, A., MARRIOTT, K., STUCKEY, P., AND YAP, R. 1996. Effectiveness of optimizing compilation for CLP(R). In *Proceedings of Joint International Conference and Symposium on Logic Programming.* MIT Press, 37–51.

LIN, Y.-J. 1988. A Parallel Implementation of Logic Programs. Ph.D. thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712.

LLOYD, J. W. 1987. *Logic Programming.* Springer-Verlag.

MARRIOT, K. AND STUCKEY, P. 1998. *Programming with Constraints: An Introduction.* The MIT Press.

MARRIOTT, K. AND STUCKEY, P. 1992. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *19th Annual ACM Conf. on Principles of Programming Languages.* ACM, 334–344.

MARRIOTT, K. AND STUCKEY, P. 1994. Approximating Interaction Between Linear Arithmetic Constraints. In *1994 International Symposium on Logic Programming.* MIT Press, 571–585.

MARTELLI, A. AND MONTANARI, U. 1982. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems 4,* 3, 258–282.

PATERSON, M. S. AND WEGMAN, M. 1978. Linear Unification. *J. of Computer and System Sciences 16,* 2, 158–167.

PEREIRA, L. M. AND PORTO, A. 1982. Selective backtracking. In *Logic Programming.* Academic Press, 107–114.

PUEBLA, G., GARCÍA DE LA BANDA, M., MARRIOTT, K., AND STUCKEY, P. 1997. Optimization of Logic Programs with Dynamic Scheduling. In *1997 International Conference on Logic Programming.* MIT Press, Cambridge, MA, 93–107.

ROBINSON, J. A. 1965. A Machine Oriented Logic Based on the Resolution Principle. *J. ACM 12,* 23 (January), 23–41.

SARASWAT, V. 1987. Compiling CP() on top of Prolog. Tech. Rep. CMU-CS-87-174, Computer Science Department, Carnegie-Mellon University, Pittsburgh. October.

UEDA, K. AND CHIKIYAMA, T. (1985). A compiler for concurrent prolog. In *2nd International Symposium on Logic Programming.* IEEE Press, Boston, 119–126.

WARREN, D. AND PEREIRA, F. C. N. 1982. An Efficient, Easily Adaptable System For Interpreting Natural Language Queries. *American Journal of Computational Linguistics 8,* 3-4, 110–122.

WARREN, R., HERMENEGILDO, M., AND DEBRAY, S. K. 1988. On the Practicality of Global Flow Analysis of Logic Programs. In *5th International Conference and Symposium on Logic Programming.* MIT Press, 684–699.

WINSBOROUGH, W. AND WAERN, A. 1988. Transparent And-Parallelism in the Presence of Shared Free variables. In *5th International Conference and Symposium on Logic Programming*. Seattle,Washington, 749–764.

Received November 1998; revised April 1999; accepted December 1999