Independence in Constraint Logic Programs

M. García de la Banda
M. Hermenegildo
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - Spain
{maria,herme}@fi.upm.es

K. Marriott
Department of Computer Science,
Monash University,
Clayton, Vic. 3168 - Australia.
marriott@cs.monash.edu.au

Abstract

Studying independence of literals, variables, and substitutions has proven very useful in the context of logic programming (LP). Here we study independence in the broader context of constraint logic programming (CLP). We show that a naive extrapolation of the LP definitions of independence to CLP is unsatisfactory (in fact, wrong) for two reasons. First, because interaction between variables through constraints is more complex than in the case of logic programming. Second, in order to ensure the efficiency of several optimizations not only must independence of the search space be considered, but also an orthogonal issue – "independence of constraint solving." 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 independence-related optimizations, from parallelism to intelligent backtracking. Sufficient conditions for independence which can be evaluated "a-priori" at run-time are also proposed. Our results suggest that independence, provided a suitable definition is chosen, is even more useful in CLP than in LP.

1 Introduction

Independence has proved to be a very useful concept in conventional logic programming (LP) as it is a necessary pre-condition for ensuring the correctness and usefulness of many important optimizations. This is exemplified in program parallelization where different notions of independence [9] and the related concept of "stability" [7] are the basis of models which incorporate Independent And-parallelism [3, 5, 9, 12, 20] as in these models the parallel execution of a set of goals in the body of a clause is ensured to be correct and efficient w.r.t. the sequential execution if the goals are proved to be independent. Independence is also the basis of optimizations such as intelligent backtracking [16] and goal reordering [21].

Here we consider independence in the more general context of constraint logic programming (CLP) [10]. CLP extends conventional logic programming by generalizing unification to constraint satisfaction. Generalizing independence to arbitrary CLP languages and constraint solvers yields new insights into independence. In particular, independence has been traditionally expressed in terms of *search space preservation*. However the generalization of the conditions for search space preservation is no longer sufficient for ensuring the efficiency of several optimizations when arbitrary CLP languages are taken into account. The reason is that even if search space is preserved, the cost of executing a set of primitive constraints may depend on the order in which those primitive constraints are considered. Thus, optimizations which vary the intended execution order established by the user, such as parallel execution, can actually cause execution to slow-down. In order to ensure efficiency, we must therefore consider an additional issue – "independence of constraint solving" – which characterizes the properties of the constraint solver behaviour when changing the order in which constraints are added. This issue has not risen previously because the standard unification algorithm is independent in this sense. However in the more general context of CLP, constraint solver independence need not hold. Here we clarify these different notions of independence: we propose various types of search independence and constraint solver independence, and show how these can be naturally combined for different applications.

The generalization should be useful since the associated optimizations performed in the context of LP appear equally applicable to the context of constraints. Indeed, the high cost of performing constraint satisfaction makes the potential performance improvements even larger. We look at three main applications. The first is and-parallelization of CLP programs. It is clear that adding constraints and running goals in parallel can dramatically improve performance. The second application is reordering of goals. This can transform a complex goal into a set of simple calculations or even simple tests. This has been shown in [13] where primitive constraints and atoms are reordered. The concepts presented here extend this optimization to allow reordering of arbitrary goals. Our third application is intelligent backtracking. This can improve efficiency by avoiding reexecution of goals (and, therefore, constraint satisfaction operations) which have no relation with the failure being handled [1]. In addition to these applications, constraint independence has another area of application which is quite specific to CLP. The idea is to decompose the single constraint solver into a number of constraint solvers each processing independent sequences of constraints. This is useful because constraint solver algorithms may not take advantage of constraint independence, and so there is potential speedup. Furthermore it may allow parallelization of the constraint solver itself.

2 Preliminaries

In this section we present the usual operational semantics of constraint logic programming (CLP) and the notation which will be used throughout the paper. We follow mainly [10], and [19]. Constraint logic programming is an extension of the logic programming paradigm in which unification is replaced by constraint solving performed over an interpreted structure not restricted to the Herbrand Universe.

Predicates in a CLP program are divided into two classes: the *primitive constraints*, *Atomic*, and the programmer-defined *atoms*, *Atom*. For simplicity we require that atoms have the form $p(x_1, .., x_n)$ where the x_i are distinct variables. Primitive constraints, however, can have terms constructed from (pre-defined) function symbols. A *literal* is an atom or a primitive constraint.

A *constraint* is a sequence of primitive constraints. However a constraint will also be considered to be the conjunction of primitive constraints and treated modulo logical equivalence. Constraints are pre-ordered by logical implication, that is $\pi \leq \pi'$ iff $\pi \Rightarrow \pi'$. We let $\exists_W \pi$ be a non-deterministic function which returns a constraint logically equivalent to $\exists V_1 \exists V_2 \cdots V_n \pi$ where variable set $W = \{V_1, \ldots, V_n\}$. We let $\overline{\exists}_W \pi$ be constraint π restricted to the variables W. That is $\overline{\exists}_W \pi$ is $\exists_{vars(\pi) \setminus W} \pi$ where function $vars$ takes a syntactic object and returns the set of (free) variables occurring in it.

A *constraint logic program (program)* is a finite set of clauses of the form $H \leftarrow \pi, B$, where the *head* H is an atom, the *guard* π is a constraint, and the *body* B is a sequence of the form $L_1, \cdots, L_n$, where each L_i is a literal. A *goal* is a (possibly empty) sequence of literals.

A *renaming* is a bijective mapping from Var to Var. We let Ren be the set of renamings and naturally extend renamings to mappings between atoms, clauses, and constraints. Syntactic objects s and s' are said to be *renamings* if there is a $\rho \in Ren$ such that $\rho(s) = s'$. The *definition of an atom A in program P with respect to variables W*, $defn_P(A, W)$, is the set of renamings of clauses in P such that each renaming has A as a head and has variables disjoint from $(W - vars(A))$.

The operational semantics of a program is in terms of "answers" to its "derivations" which are reduction sequences of "states" where a state is a tuple consisting of the current constraint, and the current literal sequence, or "goal".

A *reduction step* of state $s = \langle L : G, \pi \rangle$ for program P returns a state s' where:

1. if $L \in Atomic$ and $(L \wedge \pi)$ is satisfiable, $s' = \langle G, L : \pi \rangle$
2. if $L \in Atom$ and $(\pi' \wedge \pi)$ is satisfiable, $s' = \langle B :: G, \pi : \pi' \rangle$ and where $(L \leftarrow \pi', B) \in (defn_P(L, (vars(G) \cup vars(\pi))))$.

where ":" is the sequence constructor and "::" denotes concatenation of sequences.

A *derivation* of a state s for a program P is a finite or infinite sequence of states returned by reduction steps, starting from s. The maximal derivations of a state can be organized into a *derivation tree* in which the root of the tree is the start state and the children of a node are the states the node can reduce to. The derivation tree represents the search space for finding all answers to a state and is unique up to variable renaming. A derivation is *successful* when the last state has an empty sequence of atoms. The constraint $\overline{\exists}_s \pi$ is said to be a *partial answer* to state s if there is a derivation from s to a state with constraint π. An *answer* to state s is a partial answer corresponding to a successful derivation. We will denote the set of answers to state s by $answer(s)$, the partial answers by $partial(s)$, the derivations by $deriv(s)$ and the derivation tree by $deriv\_tree(s)$.

3 Preserving Search Space

The general, intuitive notion of independence that we would like to characterize 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 and also changing the time that it takes to compute such solutions. This time can change either because the actual number of reduction steps differs and/or because the amount of work involved in performing each of those steps differs in a significant way.

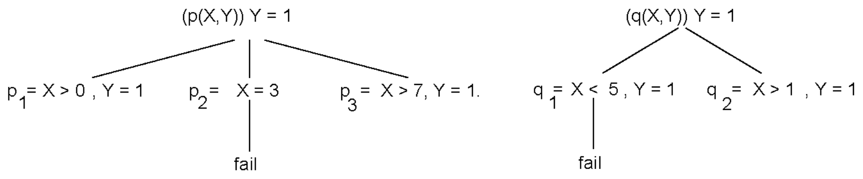
Previous work in the context of traditional Logic Programming languages [3, 5, 9] has concentrated on defining independence in terms of preservation of search space. This has been achieved by ensuring that either the goals do not share variables (*strict independence*) or if they share variables, that they do not "compete" for their bindings (*non-strict independence*)

It could be thought that these ideas might carry over trivially to CLP. However, this is not the case as the constraint systems used and their solvers can behave in ways that, from the point of view of independence, are very different from the logic programming case of equalities over first-order terms using the standard unification algorithm. There are two main issues.

First, neither strict nor non-strict independence ensure search space independence. Consider for example the state $\langle p(X) : q(Z), \{X > Y, Y > Z\}\rangle$ in a CLP($\Re$) program. Although `p` and `q` do not actually share variables just before their execution (and are thus strictly independent), it is clear that `p` and `q` can be defined in such a way that the execution of `p` prunes the search space of `q`. The second issue is that independence of search space is no longer enough to ensure independence of total execution cost: while the number of reduction steps will certainly be constant if the search space is preserved, the cost of each step may not be preserved.

In this section we will focus on the basic notions of independence which deal with search space preservation. Ensuring that the cost of each step is preserved will be the subject of Section 5.

As a final remark, note that the notions of independence previously presented for LP were generally developed with one application, program parallelization, in mind. We will show that (both for LP and for CLP) the relevant notion of a goal "affecting" another depends partly on the application domain – different applications need, in order to ensure both the correctness and the efficiency of the transformations which they perform on the program, different levels of independence. Therefore, and for generality, we will formally define the concepts of independence for CLP languages at several levels of "granularity", each of which will be shown to be "interesting" for a certain class of applications. In this sense the notions that will be presented for CLP programs, when restricted to LP, also provide new and useful concepts of independence for the LP framework.



3.1 Weak Independence

The first definition of independence 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. Consider for example 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 $\langle p(X,Y),\pi\rangle$ and $\langle q(X,Y),\pi\rangle$, with $\pi = \{Y = 1\}$[1]. Since both $(p_1 \wedge q_2)$ and $(p_3 \wedge q_2)$ are satisfiable then `p(X,Y)` and `q(X,Y)` can be considered in some sense independent for π.

Detecting this kind of independence can, in principle, be useful for performing optimizations which are based on determination of producer-consumer relationships, such as intelligent backtracking as discussed in section 6.4. Let us now formally define this level of independence which we will call *weak independence*:

Definition 1 (weak independence) *Goals g_1 and g_2 are weakly independent for constraint π iff*

$$\forall a_1 \in answer(\langle g_1,\pi\rangle).\forall a_2 \in answer(\langle g_2,\pi\rangle).(a_1 \wedge a_2) \textit{ is satisfiable}.\square$$

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.

Unfortunately, weak independence is not sufficient for ensuring search space preservation, as in the definition of weak independence only successful derivations of the goals have been considered. Therefore, we cannot ensure that an answer of a goal will not prune a failed derivation of another goal.

This can be illustrated with the previous example. Assume that we start from the state $\langle p(X,Y) : q(X,Y), \{Y = 1\}\rangle$. One of the branches of the computation will have `p` succeeding with answer substitution p_3 so that in that branch `q` will execute in the context of p_3 (i.e. as $\langle q(X,Y), p_3\rangle$). At this point it can be observed that the search space of `q` has actually been modified since q_1 would fail earlier –as soon as $X < 5$ is checked for consistency with the store by the solver– than when `q` is executed by itself.

[1]Note that, following the notation introduced in Section 2, p_1, p_3, and q_2 are answers, while p_2 and q_1 refer to the last consistent partial answers of a failed derivation.

3.2 Strong Independence

We now formally define a more restrictive concept of independence in the spirit suggested above of taking into account all partial answers which we will call *strong independence*:

Definition 2 (strong independence) *Goal g_1 is strongly independent of goal g_2 for constraint π iff*

$$\forall a \in answer(\langle g_2, \pi\rangle).\forall pa \in partial(\langle g_1, \pi\rangle).(a \wedge pa) \text{ } is \text{ } satisfiable.\Box$$

Note that while weak independence is symmetric, strong independence is not. We will now show some properties which hold for strongly independent goals. The main result is that if goal g_1 is independent of g_2 then running g_1 and g_2 in parallel can only *reduce* the size of the search space associated with the usual left to right execution of $g_1 : g_2$. We let $\#search\_sp(s)$ be the number of nodes in the derivation tree of state s.

Theorem 3.1 *If goal g_2 is strongly independent of goal g_1 for constraint π and $answers(\langle g_1, \pi\rangle) \neq \emptyset$, then*

$$\#search\_sp(\langle g_1, \pi\rangle) + \#search\_sp(\langle g_2, \pi\rangle) \leq \#search\_sp(\langle g_1 : g_2, \pi\rangle).\Box$$

This theorem[2] ensures preservation of search space w.r.t. the original left to right execution when *independently* (i.e. in different environments) executing a set of goals which satisfy the condition, with π being the original constraint store.

When reordering goals 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.[3]

Definition 3 (single solution) *A goal g is single solution for constraint π iff the state $\langle g, \pi\rangle$ has at most one successful derivation.* $\Box$

Theorem 3.2 *If goal g_2 is both strongly independent of goal g_1 and single solution for constraint π and $answers(\langle g_1, \pi\rangle) \neq \emptyset$ then*

$$\#search\_sp(\langle g_2 : g_1, \pi\rangle) \leq \#search\_sp(\langle g_1 : g_2, \pi\rangle).\Box$$

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 π. Second, the answer for g_2 (if any) will never be recomputed.

[2] Due to lack of space, all the proofs to the theorems in the paper will be omitted. They can be found in [4].

[3] This property is in general undecidable, but can be approximated. Note that the need for this property to hold in the following paragraphs is related to the preservation of the "recomputation" overhead due to the standard backtracking algorithm. Such preservation can also always be ensured by avoiding recomputation through program transformation, encapsulating goals in all-solutions predicates.

3.3 Search Independence

Finally, given the asymmetry of strong independence it is also convenient to define a symmetric notion of strong independence. We refer to this concept as *search independence*:

Definition 4 (search independence) *Goals g_1 and g_2 are search independent for constraint π iff*

$$\forall pa_1 \in partial(\langle g_1, \pi \rangle).\forall pa_2 \in partial(\langle g_2, \pi \rangle).(pa_1 \wedge pa_2) \text{ is satisfiable}.\square$$

Then, in the same spirit as Theorems 3.1 and 3.2 we can conclude:

Corollary 1 *Let $g_1, g_2 \in Goals, \pi \in Cons$ where $answers(\langle g_1, \pi \rangle) \neq \emptyset$ and $answers(\langle g_2, \pi \rangle) \neq \emptyset$. If g_1 and g_2 are search independent for π, then*

$$\#search\_sp(\langle g_1, \pi \rangle) + \#search\_sp(\langle g_2, \pi \rangle) \leq \#search\_sp(\langle g_1 : g_2, \pi \rangle).$$

$$\#search\_sp(\langle g_1, \pi \rangle) + \#search\_sp(\langle g_2, \pi \rangle) \leq \#search\_sp(\langle g_2 : g_1, \pi \rangle).\square$$

Corollary 2 *Let $g_1, g_2 \in Goals, \pi \in Cons$ where $answers(\langle g_1, \pi \rangle) \neq \emptyset$ and $answers(\langle g_2, \pi \rangle) \neq \emptyset$. If g_1 and g_2 are single solution and search independent for π, then*

$$\#search\_sp(\langle g_1 : g_2, \pi \rangle) = \#search\_sp(\langle g_2 : g_1, \pi \rangle).\square$$

As we will see in Section 6 this corollary gives us a useful tool for performing goal reordering.

4 Ensuring Search Independence "A Priori"

While compile-time detection of search independence can be based on the definitions themselves, run-time detection cannot be. This is because search independence has been defined in terms of the partial answers produced by the goals, but, in practice most applications require that run-time detection be performed just before executing the goals, and without actually having to execute them (we refer to this as "a priori" detection of independence). In order to do this (run-time) conditions for ensuring independence have to be developed which are based only on information which is readily available before executing the goals, for example in terms of the store at that point and the goals themselves. Our first approach is to define conditions which must hold for any possible partial answer:

Definition 5 (projection independence) *Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for constraint π iff*

$$\forall \pi_1, \pi_2 \in Cons\ (\pi \wedge \bar{\exists}_{\bar{x}} \pi_1) \text{ and } (\pi \wedge \bar{\exists}_{\bar{y}} \pi_2) \text{ are satisfiable} \Rightarrow$$

$$(\pi \wedge \bar{\exists}_{\bar{x}} \pi_1 \wedge \bar{\exists}_{\bar{y}} \pi_2) \text{ is satisfiable}.\square$$

Intuitively the following result holds because execution of an atom can only add constraints on local variables and the arguments of the atom.

Theorem 4.1 *Goals g_1 and g_2 are search independent for constraint π if they are projection independent for π.* $\Box$

Naive application of this sufficient condition implies testing all possible consistent constraints over the variables of each goal. Intuitively Theorem 4.1 holds iff (a) the goals do not have variables in common w.r.t. π and (b) by projecting the constraint store π over the variables of each goal we do not lose "interesting" information for the variables in each goal w.r.t. the original constraint store projected over the variables of both goals, i.e. the former store entails the latter. Therefore, a more useful characterization of projection independence can be captured by:

Theorem 4.2 *Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for constraint π iff*

$$(\bar{x} \cap \bar{y} \subseteq def(\pi)) \text{ and } (\bar{\exists}_{\bar{x}}\pi \wedge \bar{\exists}_{\bar{y}}\pi \Rightarrow \bar{\exists}_{\bar{y}\cup\bar{x}}\pi)\Box$$

Where $def(\pi)$ denotes the set of uniquely defined variables in π.

Corollary 3 *Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are search independent for constraint π if $\bar{x} \cap \bar{y} \subseteq def(\pi)$ and $\bar{\exists}_{\bar{x}}\pi \wedge \bar{\exists}_{\bar{y}}\pi \Rightarrow \bar{\exists}_{\bar{y}\cup\bar{x}}\pi$* $\Box$

For example, consider the goals $g_1(Y), g_2(Z)$ and constraint $\pi = \{Y > X, Z > X\}$. Now $\bar{\exists}_{\{Y\}}\pi = true$, $\bar{\exists}_{\{Z\}}\pi = true$, $\bar{\exists}_{\{Y,Z\}}\pi = true$. Therefore, from Corollary 3, we know that $g_1(Y), g_2(Z)$ are search independent for π.

In the Herbrand domain, for example, and due to the characteristics of the implementation of the terms and the operation of the unification algorithm, computing the projection function at run-time is immediate: the projection of the store into a set of variables is given by the objects directly pointed to by each variable. Then, in LP checking if two literals satisfy the theorem only implies ensuring that they have no variables in common. For this reason, this theorem, when considered in the context of traditional logic programs, is identical to the definition of strict independence among goals given in [9] – the most general "a priori" condition given for LP.

However, when constraint domains other than Herbrand are involved, the cost of performing a precise projection may be too high. 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.

More formally, let $def(\Pi)$ be the set of variables which Π constrains to taking a unique value. The relation $link_\Pi(x, y)$ holds for variables x and y if there is a primitive constraint π in Π such that $\{x, y\} \subseteq vars(\pi) \setminus def(\Pi)$. The relation $links_\Pi(x, y)$ is the transitive closure of $link_\Pi(x, y)$. We lift $links$ to sets of variables by defining $Links_\Pi(X, Y)$ iff $\exists x \in X.\exists y \in Y.links_\Pi(x, y)$.

Theorem 4.3 *Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are projection independent for constraint Π if $\neg Links_\Pi(\bar{x}, \bar{y})$.* $\Box$

Note that the theorem does not depend on the syntactic representation we choose for Π. 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 Π are in *normal form* if they have form $x_1 = f_1(\bar{y}) \wedge x_2 = f_2(\bar{y}) \wedge ... \wedge x_n = f_n(\bar{y}) \wedge \Pi'$ where the x_i are distinct and disjoint from the variables $\bar{y}$ and $vars(\Pi') \subseteq \bar{y}$. 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 $Links_\Pi(X,Y)$ iff $Links_{\Pi'}(vars(\psi(X)), vars(\psi(Y)))$.

The condition imposed by Theorem 4.3, although clearly sufficient, is somewhat conservative. For instance, although the goals $g_1(Y), g_2(Z)$ are search independent for $\Pi = \{Y > X, Z > X\}$, $Links_\Pi(\{Y\},\{Z\})$ holds due to the transitive closure performed when computing $links_\Pi(Y,Z)$. Thus, if projection may be efficiently performed for the particular constraint domain and solver it is better to use Theorem 4.2 to determine search independence at run time.

It is interesting to note that despite the fact that we initially considered a left-to-right execution rule, the sufficient conditions given in this section are valid independently of any computation rule. This is due to fact that these conditions are defined in terms of the information provided by the constraint store readily available before executing the goals. Thus, the conditions will remain valid no matter which computation rule will be later applied in the execution of the goals. Therefore, the results obtained in this section can be directly applied to non-deterministic CLP languages with other computation rules, such as AKL [12] or non-deterministic concurrent constraint languages in general [18].

5 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 3, this is not true in general. Modifying the order in which a sequence of primitive constraints is added to the store may have a critical influence on the time spent by the constraint solver algorithm in obtaining the answer, even if the resulting constraint is consistent. For example, consider an empty constraint store, a sequence Π_1 of primitive constraints which result in a solvable system of linear equations, and another sequence Π_2 of primitive constraints, each of them uniquely constraining one of the variables involved in the system of equations, where Π_2 is consistent with Π_1. It is clear that processing Π_1 and then Π_2 will take longer than processing Π_2 first and then Π_1. The reason is that while in the former case a relatively complex constraint solving algorithm (such as gaussian elimination) has to be applied, in the latter only simple groundness propagation is performed.

In fact, this issue is the core of the reordering application described in [13]. This is because, unlike the cost of resolving an atom which is independent of other factors, the cost of adding a primitive constraint greatly depends on the current state of the store. This issue of the variance of 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 [15, 14] – we will return to this point later. However it cannot be ignored in the context of CLP languages. For this reason, we now introduce 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 several optimizations.

Intuitively, two sequences of primitive constraints are independent of each other 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 π and π' sequences of primitive constraints. We let $cost(Solv, \pi, \pi')$ be the cost of adding the sequence π' to the solver $Solv$ after π 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 π'' is a subsequence of π', $cost(Solv, \pi, \pi'') \leq cost(Solv, \pi, \pi')$. We let $merge(\pi, \pi')$ denote the set of all mergings of the constraint sequences π and π'.

Definition 6 (sequence-solver K-independence) *Constraint sequences π' and π'' are K-independent for store π and solver $Solv$ iff $\pi' \wedge \pi'' \wedge \pi$ satisfiable implies that for all $\pi_1, \pi_2 \in merge(\pi', \pi'')$, $cost(Solv, \pi, \pi_1) - cost(Solv, \pi, \pi_2) \leq K$.* □

The intuition behind the parameterization of the definition is that the cost be bound by a constant value or function (from close to zero, to a small constant, to, for example, 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 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 (strong solver independence) *A constraint solver $Solv$ is strongly independent iff for all constraint sequences π and π', π and π' are K-independent for true and $Solv$, where K is a "small" constant value.*□

Unfortunately, many reasonable constraint solvers do not satisfy strong solver independence. In many applications a weaker notion is acceptable, namely that the solver should be solver independent for sequences which do not "interfere".

Definition 8 (weak solver independence) *A constraint solver $Solv$ is weakly independent iff for all constraint sequences π, π', and π'', if $vars(\pi') \cap vars(\pi'') \subseteq def(\pi)$, π' and π'' are K-independent for π and $Solv$, where K is a "small" constant value.*□

An even weaker independence holds if the solver can ignore independence due to uniquely defined variables.

Definition 9 (very weak solver independence) *A constraint solver $Solv$ is very weakly independent iff for all constraint sequences π, π' and π'', if $vars(\pi') \cap vars(\pi'') = \emptyset$, π' and π'' are solver K-independent for π and $Solv$, where K is a "small" constant value.*□

We claim that most reasonable constraint solvers are very weakly solver independent and many are weakly solver independent and that, therefore, the efficiency of many optimizations, such as and-parallelism, can be ensured once the adequate search space independence 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($\Re$) [11] and Prolog-III [2], constraint testing over systems of linear equations and inequations is performed using an incremental version of the Simplex algorithm. Essentially this involves incrementally recomputing a normal form for the constraint solver 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 "interface". In order to recognize such assignments or tests the solver keeps track of all variables which are constrained to a unique value. Let this constraint solver be called $Simplex$. It is easy to construct examples showing that $Simplex$ is not strongly independent. However, we do have that:

Theorem 5.1 *$Simplex$ is weakly independent.* $\square$

Obviously $Simplex$ is also very weakly independent. We believe that the reason $Simplex$ is very weakly independent is typical of many real solvers. Basically, a sufficient (and very reasonable) condition is that the number of atomic steps to add a primitive constraint π to a store Π only depends on the size of $vars(\pi)$ and the number of primitive constraints in Π in which the elements of $vars(\pi)$ appear. For weak independence to hold, the solver must additionally detect variables that are constrained to a unique value and propagate this value, as Simplex does.

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 weak 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 [15, 14].

Furthermore, if we denote by $LinUnif$ a unification algorithm belonging to the latter class, the we have that:

Theorem 5.2 *$LinUnif$ is strongly independent.* $\square$

It is interesting to point out that strong independence does not hold even within Herbrand for all solvers. For example, the cost of the original unification algorithm of Robinson [17], which is exponential in the worst case, can vary by more than a constant factor depending on reordering. It is interesting to note that the algorithm used in most practical LP 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 or they simply allow regular trees as well as terms, and also 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 LP in terms only of search independence correct.

6 Applications

As briefly mentioned in the previous sections, there are many optimizations which are based on modifying the usual sequential execution of a constraint logic program in order to improve its efficiency. Usually, the transformation requires information about (in)dependence between the goals involved in the transformation. In this section we will discuss the role of both the search and solver independence concepts introduced before in several such optimizations, which we herein call "applications."

6.1 Independent And-Parallelism

One of the aims of parallel execution schemes which exploit independent and-parallelism [3, 5, 9][4] is to run in parallel as many goals as possible while maintaining correctness and efficiency w.r.t. sequential programs. In other words, these schemes assume that the part of the model computed using a sequential execution (with a left-to-right selection rule) is the intended model of the program and that the time which the parallel execution must improve on is the time taken by the sequential system to compute this part of the model. Thus, given a goal G the idea is to execute some of the goals in G in parallel obtaining the same answer (correctness) as that obtained in its sequential execution, possibly in a shorter time but certainly not in longer time (efficiency).

It is natural to think of the extension of those ideas developed for LP to CLP. It follows from our results that in the CLP context a set of goals $g_1, \cdots, g_n$ can be allowed to be executed in parallel if:

- $\forall g_i, i : 1, \ldots, n$, g_i is strongly independent of g_j, $1 \leq j < i$ for any constraint sequence Π with which those goals can be called[5], and
- for all $d_i \in deriv(\langle g_i, \Pi\rangle)$ and for all $d_j \in deriv(\langle g_j, \Pi\rangle), i \neq j$ $cons(d_i), cons(d_j)$ are K-independent for the solver and for each possible Π, where K should somehow be less than the advantage gained by having more than one agent working in parallel (which is clearly the case if K is a small constant).

[4]This type of parallelism is complementary to or-parallelism, which is obviously always independent and can be exploited along branches of the search space of a CLP program [8].

[5]We do not require the goals to have at least one answer since although search space will not be preserved, the time spent in the parallel execution will be less or equal to that of the sequential one due to communication of failure among processors.

where *cons*(d) denotes the sequence of constraints in a derivation.

It is important to point out that although in theory the parallel execution is performed independently, i.e. without communication among parallel goals, in practice this is usually not true. The reason is that the methods used to obtain an independent execution, such as copying the store for each parallel goal and composing the results after the execution or renaming some variables and later restoring the bindings, would imply a significant overhead. Having the goals execute in different environments was enforced in order to avoid a failure due to the conjunction of a partial answer in a failing derivation of one goal and a partial answer in a successful derivation of another goal. Given the potential overhead of creating independent environments, however, and given that strong independence is already needed for ensuring correctness, it may be advantageous in practice to require a little more - search independence - since then execution in independent environments is not required.

Furthermore, if our parallel system detects that a set of goals is search independent based on the sufficient conditions provided by Theorems 4.2 or 4.3 then we can ensure that *given a weakly independent solver, efficiency is ensured.* The reason is that if one of those theorems holds for those goals, then we can ensure that all constraint sequences generated by different goals will not share variables, unless they are uniquely defined. Therefore we can ensure the second condition mentioned above, i.e. that for all constraint sequences generated by different goals and consistent with the store, those constraint sequences are K-independent for the store and the solver, where K is a small constant.

Note that the operation of the solver is actually parallelized using the scheme proposed in the sense that more than one constraint will be added to (generally independent parts of) the store.

There is of course an additional source of parallelism, complementary to the issues discussed here, related to parallelizing the actions involved in adding a single primitive constraint to the store.

6.2 Stability Detection

The notion of “stability” [12] 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 LP [6]. 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 LP concepts of independence. We believe that the results presented in this paper will be of direct application.

6.3 Reordering

In [13] an optimization based on reordering the goal $\pi \wedge g$ to $g \wedge \pi$ where π is a primitive constraint is suggested whenever π and g are strongly independent. The motivation for this is that variables in π may become uniquely defined by g, enabling the constraint π 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 reorder goals as well.

Consider the (sub-)goal $g_1 \wedge g_2$ appearing in some program, and assume that this will be called with the constraint sequences $\pi_1, \pi_2, \ldots$. This should be reordered to $g_2 \wedge g_1$ if the following two conditions are met. Firstly that changing the ordering will not increase the search space. From Theorem 3.2 a sufficient condition is that g_2 is single solution and strongly independent of g_1 and $answers(\langle g_1, \pi \rangle) \neq \emptyset$ for each π_i. Secondly there should be an improvement in the overall execution time. Thus, for each π_i and for each $d_1 \in deriv(\langle g_1, \pi_i \rangle)$ and $d_2 \in deriv(\langle g_2, \pi_i \rangle)$, $cost(Solv, \pi_i, cons(d_2) :: cons(d_1)) \leq cost(Solv, \pi_i, cons(d_1) :: cons(d_2))$ where $Solv$ is the constraint solver.

As an example of this optimizations use consider the following program `FIB` for computing the Fibonacci numbers.

```
fib(0,1).                              (FIB)
fib(1,1).
fib(N,F) ←
     N1 = N - 1, N2 = N - 2, F = F1 + F2,
     fib(N1,F1), fib(N2,F2).
```

We consider the (usual) case that this is called with constraints in which the first argument is constrained to an integer and the second argument is an unconstrained variable. In this case both recursive calls to `fib` are single solution. Furthermore, because `F`, `F1`, and `F2` are initially unconstrained, the calls to `fib` are strongly independent of `F = F1 + F2`. Thus the recursive clause body can be reordered to give the optimized program `O-FIB`.

```
fib(0,1).                              (O-FIB)
fib(1,1).
fib(N,F) ←
     N1 = N - 1, N2 = N - 2,
     fib(N1,F1), fib(N2,F2), F = F1 + F2.
```

The advantage of `O-FIB` over `FIB` is that all of the constraints in `O-FIB` are reduced to simple assignments or tests meaning that the (expensive) constraint solver is not called when `O-FIB` is executed, giving rise to substantial performance improvement.

Note that this optimization makes no sense in the case of solvers that are strongly independent. Thus it is not useful in the context of logic programming but promises to be an important optimization for CLP($\Re$).

6.4 Intelligent Backtracking

Intelligent backtracking consists in analyzing, upon unification failure, the causes of the failure and determining appropriate backtracking points that can eliminate the failure while maintaining correctness, thus avoiding unnecessary computations. The method used in LP is based both on an extended unification algorithm which keeps track of the history of the unification and performs failure analysis, and a backtrack process, which is essentially the same in all methods. One of the main decisions in this application is related to the accuracy of the unification history representation: an extremely accurate representation could be intractable, a too simple one could perform a naive backtracking at a high cost.

Let $g_1, \cdots, g_2$ be a set of goals which are weakly independent for the store π. If g_i definitely fails (i.e. it has no answers), it can be ensured that the causes of the failure are before the g_1. Therefore we can safely backtrack to the choice-point placed just before g_1, skipping all the choice-points in between.

It could be thought that, although the time saved by such optimization can be significant, the complexity of the tests needed for a run-time detection of weakly independent goals may yield a slowdown, which is clearly not the aim. However, it is important to note that, first, the traditional techniques applied to LP (i.e. keeping track of the history of the unification), are no longer valid when domains other than Herbrand are considered. Second, that maintaining a comparable structure to be able to accurately determine the causes of a failure for, for example, the constraint system based on reals with equalities and inequalities can be very complex. And third, that CLP languages are usually defined over more than one constraint system, which increases significantly the complexity of the problem. Therefore, we believe that inferring independence at compile-time or partly at compile-time and partly at run-time, and providing accurate information to the compiler so that it can specialize the program code in order to provide the appropriate links, can be a quite a useful technique for efficiently implementing intelligent backtracking in the context of CLP languages.

7 Conclusions and Future Work

We have shown how a simple extrapolation of the LP-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 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. Also, it has been shown how the concepts proposed, when applied to conventional LP, render the traditional notions and are thus a strict generalization of such notions. We believe we have in addition provided some insights into hidden assumptions related to properties of the standard unification algorithms that were made in the development of the LP concepts.

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

One clear topic for future work is to develop analyses for determining independence at compile-time. One step in this direction is the analysis for reordering given in [13]. 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 optimizations. In particular, we are in the process of developing automatic parallelization tools for CLP programs based on these ideas.

References

[1] S. Abreu, L. Pereira, and P. Codognet. Improving Backward Execution in the Andorra Family of Languages. In *1992 Joint International Conference and Symposium on Logic Programming*, pages 369–384. MIT Press, November 1992.

[2] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[3] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[4] M. Garcia de la Banda, M. Hermenegildo, and K. Marriott. Independence in constraint logic programs. Technical report, U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, November 1992.

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

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

[7] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming.* MIT Press, June 1990.

[8] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming. In *Sixth International Conference on Logic Programming*, pages 165–180, Lisbon, Portugal, June 1989. MIT Press.