(Presented at SAS'94, Namur, Belgium)

Extracting Non-Strict Independent And-Parallelism Using Sharing and Freeness Information*

Daniel Cabeza Gras and Manuel Hermenegildo

Facultad de Informática, Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - SPAIN

Abstract. Logic programming systems which exploit and-parallelism among non-deterministic goals rely on notions of independence among those goals in order to ensure certain efficiency properties. "Non-strict" independence (NSI) is a more relaxed notion than the traditional notion of "strict" independence (SI) which still ensures the relevant efficiency properties and can allow considerable more parallelism than SI. However, all compilation technology developed to date has been based on SI, because of the intrinsic complexity of exploiting NSI. This paper fills this gap by developing a technique for compile-time detection of NSI. It also proposes algorithms for combined compile-time/run-time detection, presenting novel run-time checks for this type of parallelism. The approach is based on the knowledge of certain properties regarding the run-time instantiations of program variables —sharing and freeness— for which compile-time technology is available, with new approaches being currently proposed.

1 Introduction

Several types of parallel logic programming systems and models exploit and-parallelism [**?**] among non-deterministic goals. Some examples are ROPM [**?**], AO-WAM [**?**], DDAS/Prometheus [**?**], systems based on the "Extended" Andorra Model [**?**] such as AKL [**?**], and &-Prolog [**?**] (please see their references for other related systems). All these systems rely on some notion of independence (or the related notion of "stability" [**?**]) among non-deterministic goals being run in and-parallel in order to ensure certain important efficiency properties. Two basic notions of independence are strict and non-strict independence [**?**,**?**].

Strict independence (SI) corresponds to the traditional notion of independence among goals [**?**,**?**,**?**]: Two goals g_1 and g_2 are said to be strictly independent for a substitution θ iff $\text{var}(g_1\theta) \cap \text{var}(g_2\theta) = \emptyset$, where $\text{var}(g)$ is the set of variables that appear in g. Accordingly, n goals $g_1, \ldots, g_n$ are said to be strictly independent for a substitution θ if they are pairwise strictly independent for θ. Parallelization of strictly independent goals has the property of preserving

* The research presented in this paper has been supported in part by ESPRIT projects "PARFORCE" and "ACCLAIM" and CICYT proyect TIC93-0737-C02-01.

the search space of the goals involved so that correctness and efficiency of the original program (using a left to right computation rule) are maintained and a no speed-down condition can be ensured [**?**]. A convenient characteristic of strict independence is that it is an "a-priori" condition, i.e. it can be tested at run-time ahead of the execution of the goals. Furthermore, tests for strict independence can be expressed directly in terms of groundness and independence of the variables involved. This allows relatively simple compile-time parallelization by introducing run-time tests in the program [**?**,**?**]. These tests can then be partially eliminated at compile-time by direct application of groundness and sharing (independence) information obtained from global analysis [**?**,**?**,**?**].

Non-strict independence (NSI) is a relaxation of strict independence defined as follows [**?**]:

> Consider a collection of goals $g_1, \ldots, g_n$ and a substitution θ. Consider also the set of shared variables $SH = \{v \mid \exists i, j, 1 \leq i < j \leq n, v \in (var(g_i\theta) \cap var(g_j\theta))\}$. Let θ_i be any answer substitution for $g_i\theta$. The given collection of goals is non-strictly independent for θ if the following conditions are satisfied:
> - $\forall x, y \in SH$, $\exists$ at most one $g_i\theta$ such that for a θ_i we have that $x\theta_i \not\equiv x$ or that $x \neq y$ and $x\theta_i = y\theta_i$.
> - $\forall x, y \in SH$, if $\exists$ $g_i\theta$ meeting the condition above, then $\forall g_j\theta, j > i$, such that $\{x, y\} \cap var(g_j\theta) \neq \emptyset$, g_j is a pure goal, and for all θ_j partial answer during the execution of $g_j\theta$ $x\theta_j \equiv x$ and $x\theta_i \neq y\theta_i$ if $x \neq y$.

Intuitively, the first condition of the above definition requires that at most one goal further instantiate a shared variable or alias a pair of variables. The second condition requires that any goal to the right of the one modifying the variables be pure and do not "touch" such variables in its execution. Here pure is applied to a goal that has no extra-logical builtins which are sensitive to variable instantiation.

This definition is a generalization of that originally given in [**?**]. In the case in which no information is available on the purity of goals the definition of non-strict independence can be simplified as follows:

> Consider a collection of goals $g_1, \ldots, g_n$ and a substitution θ. Consider also the set of shared variables $SH = \{v \mid \exists i, j, 1 \leq i < j \leq n, v \in (var(g_i\theta) \cap var(g_j\theta))\}$. Let θ_i be any answer substitution for $g_i\theta$. The given collection of goals is non-strictly independent for θ if $\forall x, y \in SH$, at most the rightmost $g_i\theta$ such that $x, y \in \mathrm{var}(g_i\theta)$ has an answer substitution θ_i for which $x\theta_i \not\equiv x$ or $x \neq y$ and $x\theta_i = y\theta_i$.

That is, only the rightmost goal where a shared variable occurs can further instantiate it, and only the rightmost goal where two shared variables occur can alias them. Clearly, this second definition is easier to implement, not only because no information is needed regarding the purity of the goals, which is in practice actually relatively easy to obtain, but also because no information is

needed regarding partial answers, which in general is more difficult to obtain from analyzers. This paper is focused mainly on this second definition, although some results are offered for the previous, more general, one.

Non-strict independence is clearly a more powerful notion than strict independence since strictly independent goals are always non-strictly independent. Furthermore, it still preserves the same properties as strict independence with respect to correctness and efficiency. In practice, it has wide application for example in the parallelization of programs which use difference lists, and incomplete structures in general.

Earlier studies [**?**] have suggested that coupling sharing and groundness analysis with freeness analysis could be instrumental in the task of non-strict independence detection. For concreteness, this paper focuses on a particular way of expressing sharing and freeness information, the sharing+freeness domain [**?**]. This allows a high degree of precision in the conditions involved, which are given in such a way that the implementation is straightforward. However, we believe that the ideas presented can also be used for related domains, provided that these domains give information about variable sharing and freeness.

A decision throughout our research has been to concentrate on the independence of two goals. This is convenient from a practical point of view because many parallelization algorithms work by repeatedly considering whether two goals are independent while, for example, building a dependency graph. The decision has also a sound theoretical foundation. Consider, for the case in which no information is available on the purity of goals, the following alternative definition of non-strict independence:

> Given two goals g_1 and g_2, where g_2 is to the right of g_1, and a substitution θ, consider the set of shared variables $SH = \mathrm{var}(g_1\theta) \cap \mathrm{var}(g_2\theta)$. The goals g_1 and g_2 are non-strictly independent for θ if for any answer substitution θ_1 of $g_1\theta$ and for all $x, y \in SH$, $x\theta_1$ is a variable and if $x \neq y$, $x\theta_1 \neq y\theta_1$.

Based on this definition, the definition involving n goals can be expressed as follows: $g_1, \ldots, g_n$ are non-strictly independent for a substitution θ if they are pairwise non-strictly independent for θ. Clearly, this is equivalent to the previous definition, and thus considering only pairs of goals can be done without loss of generality.

The rest of the paper proceeds as follows: Section 2 explains the particular abstract interpretation domain for which the conditions of parallelism are given, the sharing+freeness domain, and introduces a novel pictorial representation for the abstract substitutions involved. Section 3 presents the sufficient conditions proposed for compile-time detection of NSI. Section 4 deals with the combination of compile-time analyses and run-time checks for detecting NSI, presenting novel run-time checks for this type of parallelism. It also connects this method with the previously proposed techniques for the detection of strict independence. Section 5 gives some experimental results showing the speedups obtained in several programs presenting non-strict independence but no strict independence.

Section 6 discusses improvements to the compile-time analysis in order to enrich the information available for the parallelization. Finally, Sect. 7 gives the conclusions and suggests future work. For conciseness, proofs, more details on the case in which purity of goals is considered, and issues of environment separation optimization, have been omitted. They can be found in [**?**].

2 Understanding Sharing+Freeness Abstract Substitutions

The sharing+freeness abstract domain [**?**] (other related analyses for which our results may be valid include [**?**,**?**,**?**]) was proposed with the objective of obtaining at compile-time accurate variable *groundness*, *sharing*, and *freeness* information for a program, i.e., respectively, information on when a program variable will be bound to a ground term, when a set of program variables will be bound to terms with variables in common, and when a program variable will be unbound or bound only to other variables instead of to a complex term.

The abstract domain approximates this information by combining two components (in fact domains per se): the first component provides information on sharing (aliasing, independence) and groundness [**?**,**?**]; the second one provides information on freeness.

We will denote a sharing+freeness abstract substitution as a pair (sharing, freeness) as in $\widehat{\theta} = (\widehat{\theta}_{\mathrm{SH}}, \widehat{\theta}_{\mathrm{FR}})$. To distinguish abstract substitutions from concrete substitutions abstract substitutions will be represented by greek letters with a hat, the same greek letter without the hat representing a concrete substitution approximated by the abstract one. Sets will be denoted with square brackets in abstract substitutions (to distinguish them and because of the mnemonic connotations since they are to be represented in Prolog in the analyzer), and with braces in concrete substitutions (as usual). Following the standard notation, we will name the abstraction function α and the concretization function γ.

Informally, an abstract substitution in the sharing domain is a set of sets of program variables (a set of sharing sets), where sharing sets represent all *possible* sharing patterns among the program variables.

For example, given the following concrete substitution θ, $\widehat{\theta}_{\mathrm{SH}}$ is its abstraction in the sharing domain:

$$\begin{aligned} \theta &= \{\mathrm{X/f(1,a)},\ \mathrm{Y/A},\ \mathrm{Z/f(A,C,t(B))},\ \mathrm{W/[B,C]},\ \mathrm{V/D}\} \\ \widehat{\theta}_{\mathrm{SH}} &= \big[[\mathrm{YZ}]\,[\mathrm{ZW}]\,[\mathrm{V}]\big] \end{aligned}$$

On the other hand, given the following sharing abstract substitution $\widehat{\theta}_{\mathrm{SH}}$, the θ_i are concrete substitutions approximated by it. The last column in the following represents the sharing sets "active" in each concrete substitution –we say that a set $L \in \widehat{\theta}_{\mathrm{SH}}$, where $\widehat{\theta}_{\mathrm{SH}}$ is a sharing abstract substitution, is **active** in a concrete substitution $\theta \in \gamma(\widehat{\theta}_{\mathrm{SH}})$ iff L is in the abstraction of θ:

$$
\begin{array}{lll}
\widehat{\theta}_{\mathrm{SH}} = \big[[\mathrm{X}]\,[\mathrm{YZ}]\,[\mathrm{ZW}]\big] & & \\
\theta_1 \;\; = \{\mathrm{X/A},\ \mathrm{Y/f(B,1)},\ \mathrm{Z/B},\ \mathrm{W/foo}\} & \big[[\mathrm{X}]\,[\mathrm{YZ}]\big] & \\
\theta_2 \;\; = \{\mathrm{X/[\,]},\ \mathrm{Y/A},\ \mathrm{Z/[B|A]},\ \mathrm{W/t(B)}\} & \big[[\mathrm{YZ}]\,[\mathrm{ZW}]\big] & \\
\theta_3 \;\; = \{\mathrm{X/t(0,1)},\ \mathrm{Y/atom},\ \mathrm{Z/A},\ \mathrm{W/A}\} & \big[[\mathrm{ZW}]\big] &
\end{array}
$$

The component described above is essentially the abstract domain of Jacobs and Langen [**?**].

An abstract substitution in the freeness domain is a set of program variables (those that are known to be free).

It is important to point out that sharing information is not independent of freeness information, since known freeness of certain variables restricts the allowable combinations of sharing sets. The possible combinations of sharing sets a sharing+freeness abstract substitution $\widehat{\theta}$ represents are the subsets of the sharing component (the $S \in \wp(\widehat{\theta}_{\mathrm{SH}})$) that have one and only one sharing set including each variable in the freeness component ($\forall\, v \in \widehat{\theta}_{\mathrm{FR}}\ \dot{\exists}\, L \in S\ \ v \in L$).

The point above regarding sharing+freeness abstract substitutions, which is of great practical importance, may still be difficult to understand in the terms given so far. It is hoped that with the aid of the pictorial representation to be presented in the following section these issues will be greatly clarified.

2.1 Pictorial Representation of Substitutions

We have chosen a pictorial representation of substitutions in order to make it easier to understand abstract substitutions in the sharing+freeness domain and to follow the discussions and examples throughout the text. The idea of the pictures is to make the large amount of information contained in these abstract substitutions more explicit. Figure 1 illustrates the different types of objects used in this representation.

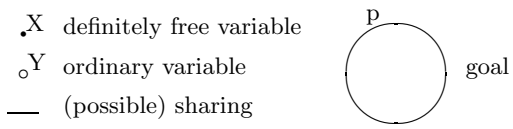


Fig. 1. Types of objects in our pictorial representation.

As mentioned before, an abstract sharing+freeness substitution is a compact representation of a finite number of possible sharing+freeness situations in the concrete domain. To reflect this a given sharing+freeness abstract substitution can be represented with a finite number of figures, each figure having the same freeness information (which is definite) but representing the different alternative coverings of free variables by the sharing sets.

As the sharing+freeness abstract substitutions give information in terms of program variables, only these variables appear in the figures. Variables in the

freeness component are represented by dots, the rest by circles. The sharing patterns are represented by lines connecting all the variables of the corresponding sharing set. If the sharing set has only one variable, and this variable is not in the freeness component, it is represented as a short line coming out of the variable. Note that all sharing sets that contain no free variables (including those just mentioned involving only one variable), may be either active or inactive in a concrete substitution, since they represent only "possible" sharing. Rather than having multiple figures for all the cases involved, all such sets are represented explicitly by lines in a given picture, under the assumption that those lines may or may not be present. Note also that lines connecting free variables on the other hand represent necessarily active sharing sets and cannot be removed.

The number of lines coming out of a circle represents the number of sharing sets containing the corresponding variable. However, multiple lines coming out of dots (free variables) all correspond to the same sharing set, since free variables must be in one and only one active sharing set (this is done to simplify the drawings). If no line comes out of a given dot this represents a sharing set containing only this variable. Note that a circle connected to one or more lines can in fact represent a free variable, since the freeness component names only the variables that are definitely known to be free. On the other hand, an isolated circle represents always a ground variable, since the variable is not a member of any sharing set. The resulting pictures are hypergraphs, since the edges connect an arbitrary number of vertices.

A goal is represented like a set in a Venn Diagram, the variables in the set being the goal variables. When we represent two goals, the first one (in the Prolog textual order) is to the left and the second one to the right, and the variables present in both goals are placed in the "intersection".

Figure 2 shows several examples representing in one or more pictures abstract substitutions.

3 Conditions for Non-Strict Independence with Respect to the Information from Sharing+Freeness Analysis

The definition of non-strict independence that we use (for two goals) is given in terms of the substitutions before and after the execution of the goal to the left, i.e. of its call and answer substitutions. Correspondingly, in the abstract domain we will consider that goal's abstract call and abstract answer substitutions.

3.1 Conditions Disregarding Purity of Goals

As mentioned in the introduction we will consider the parallelization of pairs of goals. First let us state the conditions without purity information.

Let p and q be two goals, where q is to the right of p. Also let $\widehat{\beta}$ and $\widehat{\psi}$ be the abstract call and answer substitutions for p. So the situation is $\{\widehat{\beta}\}$ p $\{\widehat{\psi}\}$ ... q.

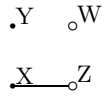
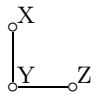
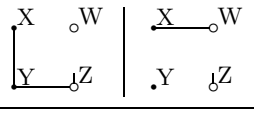
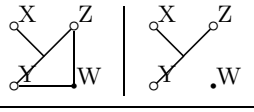
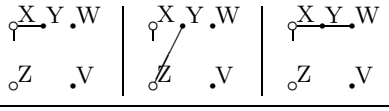
Variables / Abstract substitution	Representation
{X, Y, Z, W} / $\big([[\mathrm{Y}]\,[\mathrm{XZ}]], [\mathrm{XY}]\big)$	
{X, Y, Z} / $\big([[\mathrm{XY}]\,[\mathrm{YZ}]], []\big)$	
{X, Y, Z, W} / $\big([[\mathrm{XYZ}]\,[\mathrm{XW}]\,[\mathrm{Y}]\,[\mathrm{Z}]], [\mathrm{XY}]\big)$	
{X, Y, Z, W} / $\big([[\mathrm{XYZ}]\,[\mathrm{YZW}]\,[\mathrm{W}]], [\mathrm{W}]\big)$	
{X, Y, Z, W, V}/ $\big([[\mathrm{X}]\,[\mathrm{XY}]\,[\mathrm{YZ}]\,[\mathrm{W}]\,[\mathrm{XYW}]\,[\mathrm{V}]], [\mathrm{YWV}]\big)$	

Fig. 2. Examples of representation of abstract substitutions

We define the sets:

$$\mathrm{S(p)} = \big\{L \in \widehat{\beta}_{\mathrm{SH}} \mid L \cap \mathrm{var(p)} \neq \emptyset\big\}$$
$$\widehat{\mathrm{SH}} \;= \mathrm{S(p)} \cap \mathrm{S(q)} = \big\{L \in \widehat{\beta}_{\mathrm{SH}} \mid L \cap \mathrm{var(p)} \neq \emptyset \;\wedge\; L \cap \mathrm{var(q)} \neq \emptyset\big\}$$

That is, S(p) is the set of all sharing sets of $\widehat{\beta}_{\mathrm{SH}}$ that contain a variable from p, and $\widehat{\mathrm{SH}}$ is the set of all sharing sets of $\widehat{\beta}_{\mathrm{SH}}$ that contain variables from p and from q (that is, the sharing sets that, if are active, contain run-time shared variables).

The following are our conditions for non-strict independence between p and q:

C1 $\;\forall\, L \in \widehat{\mathrm{SH}}\;\; L \cap \widehat{\psi}_{\mathrm{FR}} \neq \emptyset$

C2 $\;\neg\,(\exists\, N_1 ... N_k \in \mathrm{S(p)}\;\exists\, L \in \widehat{\psi}_{\mathrm{SH}}\;\; (L = \bigcup_{i=1}^{k} N_i) \;\wedge\; N_1, N_2 \in \widehat{\mathrm{SH}}$
$\wedge\, \forall\, i, j\;\; 1 \leq i < j \leq k\;\; N_i \cap N_j \cap \widehat{\beta}_{\mathrm{FR}} = \emptyset)$

Condition C1 deals with preserving freeness of shared variables[1]. By checking that all sharing sets of $\widehat{\mathrm{SH}}$ have a free variable in the abstract answer substitution $\widehat{\psi}$, it is ensured that no run-time shared variable is further instantiated. Note that if there is more than one free variable in a sharing set, and one of them remains free, the other necessarily remain also free, since all coincide at run-time when the set is active.

[1] We would like to thank M. Bruynooghe for suggesting improvements to our first C1.

Condition C2 is needed to preserve independence of shared variables: $N_1 ... N_k$ are sharing sets that p can unite (thus they come from S(p)) to derive the sharing set L of the abstract answer substitution, and at least two sharing sets contain shared variables (we can always name them N_1 and N_2). Furthermore, no two sharing sets N_i, N_j contain the same free variable, since otherwise they cannot be both active in one concrete substitution, making the union impossible. This also ensures, given that the first condition is met, that N_1 and N_2 have different shared variables. Intuitively it can be seen that if C1 and ¬C2 holds, p can possibly bind the two independent shared variables.

Figure 3 shows some situations where either C1 or C2 do not hold. The sharings drawn with thick lines are the faulty ones, i.e. for C1, the Ls that have no variables in $\widehat{\psi}_{\mathrm{FR}}$, and for C2, N_1 and N_2 in $\widehat{\beta}$ and L in $\widehat{\psi}$.

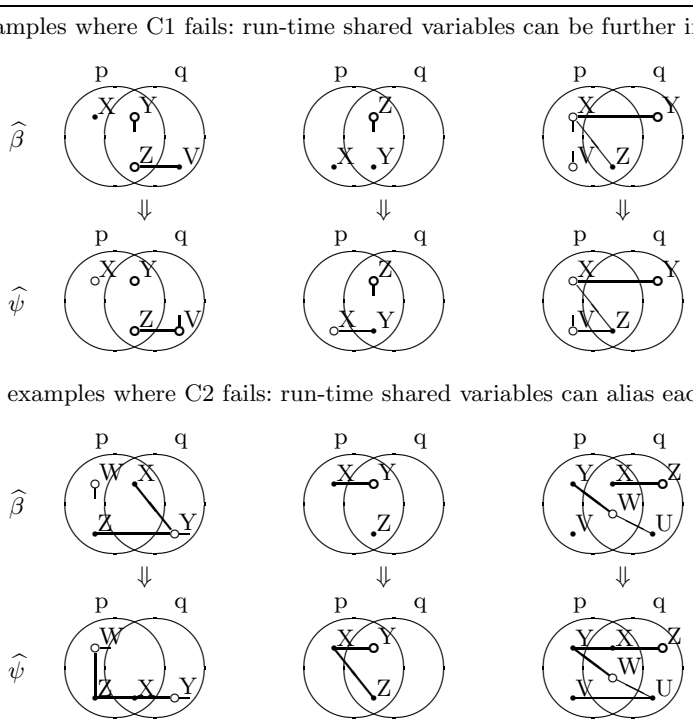


Fig. 3. Situations where the conditions do not hold, and thus the goals are possibly not NSI

3.2 Conditions Considering Purity Information

This section relies on the assumption that we have purity information, and also that we can compute the least upper bound of the abstractions of the partial answers of a goal.

If we examine the conditions stated in the previous section, we can see that only the behavior of the first goal p is considered. But if we know that q is pure, the conditions can be further relaxed.

Let us now define our conditions for non-strict independence when q is pure. Let $\widehat{\beta}$ and $\widehat{\psi}$ be the abstract call and answer substitutions for p, and let $\widehat{\phi}$ be the least upper bound of the abstractions of the partial answers of q when called with $\widehat{\beta}$ as the abstract call substitution. The following are our conditions for non-strict independence between p and q in this case:

$$\begin{array}{ll} \text{C1}' & \forall\, L \in \widehat{\text{SH}}\ \ L \cap \widehat{\psi}_{\text{FR}} \neq \emptyset \ \vee\ L \cap \widehat{\phi}_{\text{FR}} \neq \emptyset \\ \text{C2}' & \neg\,(\exists\, N_1 ... N_k \in \text{S(p)}\ \exists\, L \in \widehat{\psi}_{\text{SH}}\ \ (L = \bigcup_{i=1}^{k} N_i) \wedge N_1, N_2 \in \widehat{\text{SH}} \\ & \qquad \wedge \forall\, i,j\ 1 \leq i < j \leq k\ N_i \cap N_j \cap \widehat{\beta}_{\text{FR}} = \emptyset \wedge N_1 \cap \widehat{\phi}_{\text{FR}} = N_2 \cap \widehat{\phi}_{\text{FR}} = \emptyset) \end{array}$$

Condition C1′ differs from C1 in that it allows p to further instantiate a shared variable, provided that this variable is not touched by q (q does not further instantiate it under $\widehat{\beta}$, so it does not mind whether the variable is free or not). Condition C2′ now says that the union of N_1 and N_2 is legal if either of the shared variables in them is not touched by q (note that only if q further instantiates the two variables it can possibly be affected by these bindings).

4 Run-Time Checks for Non-Strict Independence

In the previous sections we have proposed conditions to be checked at compile-time in order to decide whether to run two goals in parallel. However, even if these conditions do not hold, we may yet try to execute them in parallel, provided that some a priori run-time checks succeed.

The purpose of the run-time checks is to ensure that goals will not be run in parallel when there is no non-strict independence, while allowing parallel execution in as many cases as possible when non-strict independence is present. This fact will be determined from the combination of compile-time analysis and the success of the run-time checks previous to the execution of the goals. Note that this is meaningful because the sharing component represents possible, not definite sharing sets. Thus we may want to generate a test that determines in which particular case we are, when at least one case allows parallelization, but the others may not.

Due to the complexity of the special case when the second goal is pure, here we will only consider the general case (see [**?**] for conditions for the case in which purity is taken into account). Let us analyze what to do when each of the conditions of the general case is violated.

4.1 Condition C1 Violated

$$[\exists\, L \in \widehat{\text{SH}}\ \ L \cap \widehat{\psi}_{\text{FR}} = \emptyset]$$

In this case we need run-time checks to ensure that the sharing sets $L \in \widehat{\text{SH}}$ not obeying C1 ("illegal sharing sets") are not active. But, if the rest of the

sharing sets in $\widehat{\beta}_{\mathrm{SH}}$ cannot cover all the free variables of $\widehat{\beta}_{\mathrm{FR}}$ without overlapping, it is impossible for all the illegal sharing sets to be inactive, so the goals are definitely not NSI. Otherwise, we must try to generate the least number of checks which covers every illegal sharing set without affecting the legal ones (to preserve parallelism in valid situations).

There are several checks that can be used to prevent the illegal sharing sets from being active. The order in what must be tried is the following:

- If there exists a variable X such that it appears only in illegal sharing sets, then the check `ground(X)` ("X is bound to a ground term") covers those illegal sharing sets containing X.
- Suppose that there exists a variable X and a list $\mathcal{F}$ of free variables from $\widehat{\beta}_{\mathrm{FR}}$ such that, for the sharing sets containing X, illegal ones do not contain variables of $\mathcal{F}$, and legal ones contain at least one. Then the check `allvars(X,`$\mathcal{F}$`)` ("every variable in X is in the list $\mathcal{F}$") covers all the illegal sharing sets containing X, and only those. In fact, the check `ground(X)` above is a special case of this when $\mathcal{F} = [\,]$. Note that if $\mathrm{X} \in \mathrm{var(p)} \cap \mathrm{var(q)}$ then we always are in this case, since all sharing sets containing X are in $\widehat{\mathrm{SH}}$, so the ones that are legal contain free variables that remain free after executing p, and those that are illegal do not.
- If there exist two variables X and Y such that all sharing sets containing both are illegal, then the check `indep(X,Y)` ("X and Y do not share variables") covers those illegal sharing sets.
- For each of the remaining illegal sharing sets, we choose two variables X and Y which are members of it, such that $\mathrm{X} \in \mathrm{var(p)}$ and $\mathrm{Y} \in \mathrm{var(q)}$. Note that the sharing sets in $\widehat{\mathrm{SH}}$ have a variable in both var(p) and var(q) or have one variable in var(p) and another variable in var(q). And, since the illegal sharing sets are in $\widehat{\mathrm{SH}}$, if they cannot be covered by the `allvars/2` check then they are in this case. Furthermore, the legal sharing sets that contain both X and Y are for this very reason also in $\widehat{\mathrm{SH}}$, so they have free variables that remain free after executing p. Let $\mathcal{F}$ be the set of these free variables. Then the check `sharedvars(X,Y,`$\mathcal{F}$`)` ("every variable shared by X and Y is in the list of variables $\mathcal{F}$") covers all the illegal sharing sets containing X and Y, and only those. Also, the check `indep(X,Y)` is a special case of this when $\mathcal{F} = [\,]$.

Figure 4 gives examples showing how the checks restrict the possible sharing sets.

4.2 Condition C2 Violated

$$[\exists\, N_1 ... N_k \in \mathrm{S(p)}\ \exists\, L \in \widehat{\psi}_{\mathrm{SH}}\ \ (L = \textstyle\bigcup_{i=1}^{k} N_i) \\ \wedge\ N_1, N_2 \in \widehat{\mathrm{SH}} \wedge \forall\, i, j\ \ 1 \leq i < j \leq k\ \ \ N_i \cap N_j \cap \widehat{\beta}_{\mathrm{FR}} = \emptyset]$$

Once the checks for C1 have been computed, and taking into account only the sharing sets not rejected by these checks, the second condition is treated.

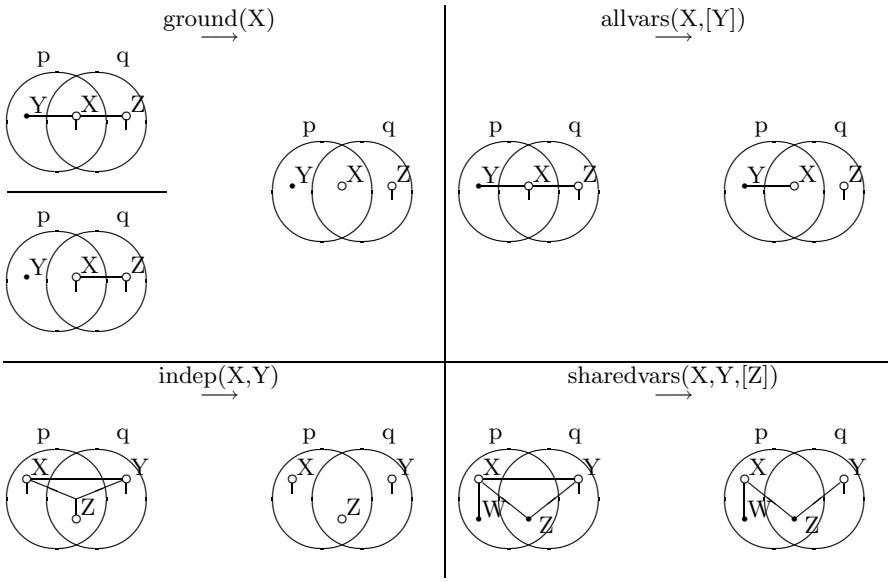


Fig. 4. Example applications of the four checks

Now, for each L in the above formula, we compute the different groups of $N_1 ... N_k$ that p can unite to give the sharing set L, without taking into account the number of sharing sets N_i that are in $\widehat{\mathrm{SH}}$. The groups that have more than one sharing set in $\widehat{\mathrm{SH}}$ are the "illegal" groups. If there are no legal groups, and L is necessarily active in $\widehat{\psi}$ (this is so if L contains free variables that do not appear in other sharing sets of $\widehat{\psi}_{\mathrm{SH}}$), then necessarily p binds shared variables, so the goals are definitely not NSI. Otherwise, we need checks as for the first condition, now ensuring that at least one sharing set of each illegal group is not active, without affecting if possible sharing sets of the legal groups.

For example, suppose we are trying to parallelize the goal "`p(X,Y,Z,U), q(X,Y,W,V)`" and the abstract call and answer substitutions for `p(X,Y,Z,U)` are $\widehat{\beta} = \big([[\mathrm{X}]\,[\mathrm{XZ}]\,[\mathrm{Y}]\,[\mathrm{Z}]\,[\mathrm{ZW}]\,[\mathrm{U}]\,[\mathrm{UW}]\,[\mathrm{WV}]], [\mathrm{YUV}]\big)$ and $\widehat{\psi} = \big[[\mathrm{X}]\,[\mathrm{YU}]\,[\mathrm{UW}]\,[\mathrm{WV}]], [\mathrm{YV}]\big)$. We have that $\widehat{\mathrm{SH}} = \big[[\mathrm{X}]\,[\mathrm{XZ}]\,[\mathrm{Y}]\,[\mathrm{ZW}]\,[\mathrm{UW}]\big]$, and the illegal sharing sets for the first condition are [X], [XZ], [ZW] and [UW]. The check `ground(X)` covers the first two, and the check `allvars(W,[V])` the last two (without affecting other sharing sets). The second condition holds, so we are ready to parallelize the two goals, the result being:

```
(ground(X),allvars(W,[V]) -> p(X,Y,Z,U)&q(X,Y,W,V);p(X,Y,Z,U),q(X,Y,W,V))
```

where "`A -> B; C`" is the prolog *if-then-else* and "`&`" is the (unconditional) parallel operator. Figure 5 shows the restriction of the possible sharing sets made by the checks, and how this restriction make the goals non-strict independent.

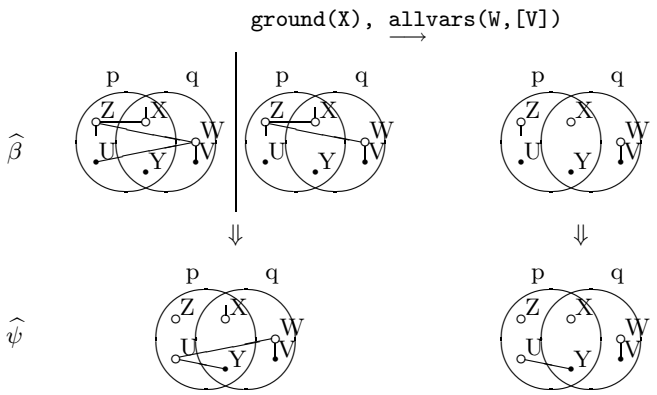


Fig. 5. Restriction of the possible sharing sets by the checks

4.3 Run-Time Checks and Strict Independence

It is worth pointing out that if no information is obtained from the analysis (or no analysis is performed), and thus the abstract substitutions are $\top$, the run-time checks computed by the method presented here exactly correspond to the conditions traditionally generated for strict independence (shared program variables ground, other program variables independent, see e.g. [**?**] for more information). This is correct, since in absence of analysis information only strict independence is possible, and shows that the method presented is a strict generalization of the techniques which have been previously proposed for the detection of strict independence.

It can be easily shown how the tests reduce to those for strict independence: since there are no free variables in the abstract substitutions, every sharing set of $\widehat{\mathrm{SH}}$ is illegal with respect to the first condition. These sharing sets contain a shared program variable (and are covered by a `ground/1` check on each) or program variables of both goals (covered by an `indep/2` check on every pair).

For example, if we have a goal "`p(X,Y)&q(Y,Z)`" with $\widehat{\beta} = \big(\big[[\mathrm{X}]\,[\mathrm{Y}]\,[\mathrm{Z}]\,[\mathrm{XY}]\,[\mathrm{XZ}]\,[\mathrm{YZ}]\,[\mathrm{XYZ}]\big], [\,]\big)$ (i.e. $\top$, equivalent to no information), then we have $\widehat{\mathrm{SH}} = \big[[\mathrm{Y}]\,[\mathrm{XY}]\,[\mathrm{XZ}]\,[\mathrm{YZ}]\,[\mathrm{XYZ}]\big]$. The check `ground(Y)` covers all the illegal sharing sets except [XZ], which is covered in turn by the check `indep(X,Z)`. Figure 6 depicts how the checks restrict the possible sharing sets.

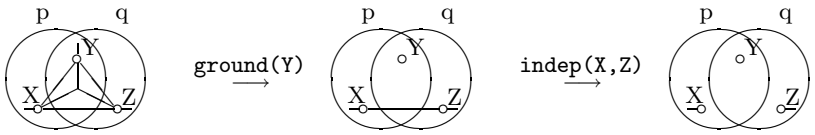


Fig. 6. Restriction of the possible sharing sets performed by the checks

Also, in the presence of sharing+freeness abstract information, the tests made with this method are equivalent or better than the traditional tests simplified with this information, even if only strict independence is present. As an example, let us study the case of the goal "`p(X,V,W)&q(Y,Z)`" with $\widehat{\beta} = \big([[V]\,[VX]\,[Y]\,[XY]\,[Z]\,[XZW]\,[W]], [V]\big)$ (see Fig. 7). The traditional test for strict independence would be `indep(V,Y), indep(X,Y), indep(W,Y), indep(V,Z), indep(X,Z), indep(W,Z)` (perhaps written as `indep([V,X,W],[Y,Z])`). With the analysis information above, is simple to deduce that the tests `indep(V,Y)`, `indep(W,Y)` and `indep(V,Z)` are not needed. Not so obvious is to deduce that one of the test `indep(X,Z)` or `indep(W,Z)` can also be eliminated. So, in this latter case we come up with the simplified test `indep(X,Y), indep(X,Z)`, or `indep(X,[Y,Z])`.

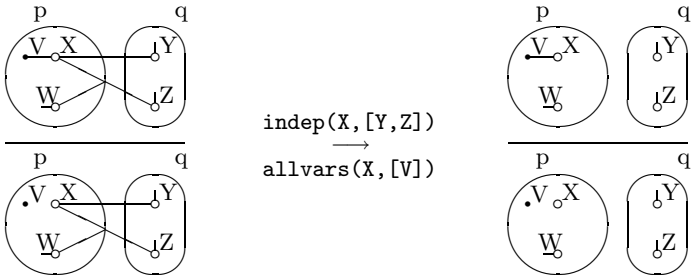


Fig. 7. Restriction of the possible sharing sets performed by either check

On the other hand, applying the method presented here, we have that $\widehat{\mathrm{SH}} = \big[[XY]\,[XZW]\big]$. Both sharing sets are illegal, since they do not contain free variables. The legal sharing set that contains X contains also the free variable V, and the two illegal sharing sets contain X but not this free variable, so `allvars(X,[V])` ensures that the illegal sharing sets are inactive, without affecting any legal sharing set. This test is clearly cheaper than the other, since it only needs to traverse X, whereas the other needs to traverse also Y and Z (in the worst case).

5 Some Experimental Results

We have measured the speedups obtained using the techniques presented for a number of programs that have NSI but no SI. The programs were automatically parallelized with our parallelizing compiler. This compiler is a modification of the &-Prolog compiler, which was originally designed to exploit strict independence. New annotator and dependency analysis modules were added which implement the techniques presented so far. Only unconditional parallelism was used (i.e. no run-time checks were generated). The programs where then executed using from 1 to 10 processors on a Sequent Symmetry on the &-Prolog system [**?**],

an efficient parallel implementation of full Prolog that can exploit independent and-parallelism among non-deterministic goals.

The results are given in Table 1. Speedups are relative to the sequential execution on one processor. The performance of &-Prolog on one processor, even when running parallelized programs, is about 95% of the performance of the sequential system (SICStus Prolog [**?**]) on which it is based, itself one of the most popular Prolog systems. Thus, we argue, the speedups obtained are meaningful and useful, and we believe that the results obtained are quite encouraging. The differences between the sequential execution and the execution of the parallelized program on one processor is most due to the environment separation issue, mentioned in the introduction (see [**?**]).

A description of the programs used follows: the **array2list** program is a subroutine of the SICStus prolog "arrays.pl" library. It translates an extendable array into a list of index-element pairs. The input array used to measure the speedups had 2048 elements. The **flatten** program is a subroutine that flattens a list of lists of any complexity into a plain list. The speedups were measured with an input list of 987 elements with recursive "depth" of seven. The **hanoi_dl** program is the well-known benchmark that computes the solution of the towers of Hanoi problem, but programmed with difference lists. It was run for 13 rings. The **qsort** program is the sorting algorithm quicksort using difference lists. The speedups were measured sorting a list of 300 elements. Finally, the **sparse** program is a subroutine that transforms a binary matrix (in the form of list of lists) into a list of coordinates of the positive elements, i.e. a sparse representation. It was run with an input matrix of 32×128 elements, with 256 positive elements.

Table 1. Speedups of several programs with NSI

Bench	# of processors									
	1	2	3	4	5	6	7	8	9	10
array2list	0.78	1.54	2.34	3.09	3.82	4.64	5.41	5.90	6.50	7.22
flatten	0.54	1.07	1.61	2.07	2.52	3.05	3.62	4.14	4.46	4.83
hanoi_dl	0.56	1.13	1.68	2.25	2.73	3.23	3.70	4.34	4.84	5.25
qsort	0.91	1.65	2.20	2.53	2.75	2.86	3.00	3.14	3.30	3.33
sparse	0.99	1.92	2.79	3.68	4.50	5.06	5.78	6.75	8.10	8.26

6 Towards an Improved Analysis for Non-Strict Independence

We have so far presented a method for detecting non-strict independence from the information provided by a straightforward analysis based on the Sharing+ Freeness domain. In light of this method we were able to understand more clearly in what way the analysis itself can be improved to increment the amount of parallelism that can be exploited automatically.

A first way to do this is by combining Sharing+Freeness with other analyses that can improve the accuracy of the sharing and freeness information. A class of such analyses includes those that use linearity, such as the Asub domain [**?**] (among others). In fact, this idea has already been incorporated in our system by using the techniques described in [**?**], and the results are used by the non-strict independence parallelizing compiler by simply focusing only on the Sharing+Freeness part. However, the improvement that can be obtained by these means is limited, as long as the sharing and freeness information is restricted to program variables.

A better improvement could be achieved by gaining access to information inside the terms to which program variables are bound at run-time, in order to check the possible instantiations of free variables inside these terms. To achieve this goal, sharing and freeness could be integrated (by using the techniques of [**?**] or [**?**]) with other analyses, like the depth-k [**?**] domain, or, even better, "pattern" [**?**] or any other recursive type analysis (see [**?**]), at least for lists. This would allow dealing, for example, with lists of free variables. These alternatives will be studied in future work. However, note that the approach presented here is still valid directly or with very slight modifications for these more sophisticated types of analyses.

7 Conclusions

We have presented several techniques for achieving the compile-time detection of non-strict independence. The proposed techniques are based on the availability of certain information about run-time instantiations of program variables –sharing and freeness– for which compile-time technology is available, and for the inference of which new approaches are being currently proposed. We have also presented techniques for combined compile-time/run-time detection of NSI, proposing new kinds of run-time checks for this type of parallelism as well as the algorithms for implementing such checks. Experimental results showing the speedups found in some programs presenting NSI have also been given. The results were obtained by integrating the algorithm that detects non-strict independence (and others needed to exploit this kind of independence) in our parallelizing compiler, that already included a sharing+freeness analyzer, obtaining a complete compile-time parallelizer capable of detecting non-strict independence. We find that the results are encouraging.

We are also planning on looking, in the light of the techniques developed, to more sophisticated abstract analyses that may provide more accurate information, in order to increment the amount of parallelism exploitable automatically.

References