facultad de informática

universidad politécnica de madrid

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

Daniel Cabeza Gras

Manuel Hermenegildo

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

Authors

Daniel Cabeza Gras
dcabeza@dia.fi.upm.es
Manuel Hermenegildo
herme@fi.upm.es

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

Keywords

Parallel Execution of Logic Programs, Compilation Techniques, Generation of Annotations for Parallelism, Abstract Interpretation, Non-strict Independent And-Parallelism.

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 is related to the fact that NSI cannot be determined "a priori" as SI. This paper fills this gap by developing a technique for compile-time detection and annotation of NSI. It also proposes algorithms for combined compile-time/run-time detection, presenting novel run-time checks for this type of parallelism. Also, a transformation procedure to eliminate shared variables among parallel goals is presented, aimed at performing as much work as possible at compile-time. 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. Thus, the paper does not deal with the analysis itself, but rather with how the analysis results can be used to parallelize programs.

Contents

1 Introduction

Several types of parallel logic programming systems and models exploit and-parallelism [5] among non-deterministic goals. Some examples are PEPsys [27], ROPM [21], AO-WAM [9], DDAS/Prometheus [23], systems based on the "Extended" Andorra Model [26] such as AKL [16], and &-Prolog [11] (please see their references for other related systems). All these systems rely on some notion of independence (or the related notion of "stability" [10]) 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 [12, 13].

Strict *goal* independence corresponds to the traditional notion of independence among goals [5, 7, 11]: Two goals g_1 and g_2 are said to be strictly independent for a substitution θ iff $\text{var}(g_1\theta) \cap \text{var}(g_2\theta) = \emptyset$; 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 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 [12]. 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 [7, 19]. These tests can then be partially eliminated at compile-time by direct application of groundness and sharing (independence) information obtained from global analysis [15, 18, 3].

Non-strict independence is a relaxation of strict independence traditionally defined as follows: given a collection of goals $g_1, \ldots, g_n$ and a substitution θ, let $SH = \{v \mid \exists i,j\ 1 \leq i < j \leq n, v \in \text{var}(g_i\theta) \cap \text{var}(g_j\theta)\}$, let θ_i be any answer substitution for $g_i\theta$, then $g_1, \ldots, g_n$ are non-strictly independent for θ iff $\forall v \in SH$, at most the rightmost g_i such that $v \in \text{var}(g_i\theta)$ binds v to a non-variable term, and if $\text{var}(g_i\theta)$ contains more than one variable of SH, say $x_1, \ldots, x_k$, then $x_1\theta_i, \ldots, x_k\theta_i$ are *strictly independent* [13]. 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. In fact,

studies of amounts of ideal parallelism in logic programs suggest that there is a potential for large speedups from the exploitation of non-strict independence [23]. However, this potential remains untapped from the point of view of automatic parallelization. This is due to two factors. The first one is that non-strict independence is not an "a priori" condition, i.e. it cannot be expressed simply in terms of run-time tests (without running the goals). Thus, run-time detection by itself is ruled out. Unfortunately, compile-time detection is complicated by the fact that non-strict independence is not directly expressed in the same terms as the properties which are usually determined from global analysis.

Earlier studies [12] have suggested that coupling sharing and groundness analysis with freeness analysis could be instrumental in the task of non-strict independence detection. This has been one of the motivations behind the development of analyzers capable of inferring these three types of information [4, 6, 25, 20, 8]. However, there still remained a semantic gap between the availability of that information and actually being able to reason about the non-strict independence of a set of goals. This paper attempts to fill this gap. It aims to develop concrete techniques for determining non-strict independence at compile-time. For concreteness, it focuses on a concrete way of expressing sharing and freeness information, the sharing+freeness domain [20]. 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.

One design decision throughout our research was to concentrate on the parallelization of two goals or two sets of goals (containing either sequential or parallel constructs). This is convenient from a practical point of view because many parallelization algorithms work by repeatedly considering whether two goals or sequences are independent while, for example, building a dependency graph. The algorithms described in this paper are directly aimed at answering such questions for the case of non-strict independence. The decision of considering the parallelization of pairs of goals has also a sound theoretical foundation. Consider 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 $SH = \mathrm{var}(g_1\theta) \cap \mathrm{var}(g_2\theta)$. Then, g_1 and g_2 are non-strictly independent for θ iff for any answer substitution θ_1 of $g_1\theta$ and for all v,$w \in SH$, $v\theta_1$ is a variable and $v \neq w \rightarrow v\theta_1 \neq w\theta_1$. Based on this definition, the definition involving n goals can be expressed as: $g_1, \ldots, g_n$ are non-strictly independent for a substitution θ if they are pairwise non-strictly independent for θ. Clearly, this is equivalent to the standard definition, and thus considering only pairs of goals can

always 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 develops an efficient algorithm for performing combined compile-time/run-time renaming of variables, which is needed for the parallel execution of non-strictly independent goals. Section 6 illustrates the techniques proposed by using them to parallelize of a concrete program. Section 7 gives some experimental results showing the speedups obtained in several programs presenting non-strict independence but no strict independence. Section 8 treats the parallelization of goals when additional information about purity of predicates is available, and proposes new approaches related to compile-time analysis in order to improve the information required for the parallelization techniques. Finally, section 9 gives the conclusions and suggests future work.

2 Understanding Sharing+Freeness Abstract Substitutions

The sharing+freeness abstract domain [20] (other related analyses for which our results may be valid include [4, 6, 25, 8]) 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 [15, 18]; the second one provides information on freeness. More precisely, $D_\alpha \subset \bot \cup \wp(\wp(Pvar)) \times \wp(Pvar)$, where $Pvar$ is the set of all program variables in the current clause. It is an inclusion and not an equality because abstract substitutions in $\wp(\wp(Pvar)) \times \wp(Pvar)$ whose concretization would be empty are not considered (they are represented by $\bot$ —bottom).

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.

More formally, let us define a (concrete) substitution in a clause as a mapping from the set of program variables in that clause ($Pvar$) to terms that can be formed from the constants and the functors in the given program and in the query and an infinite set of variables Var such that $Pvar \cap Var = \emptyset$. In this way we consider only idempotent substitutions. The set of all concrete substitutions will be denoted as $Subst$.

The function Occ: $Subst \times Var \rightarrow \wp(Pvar)$ is defined such that

$$Occ(\theta, V) = \{X \in \mathrm{dom}(\theta) \mid V \in \mathrm{var}(X\theta)\}$$

where $t\theta$ denotes the instantiation of a term t under a substitution θ, $\mathrm{var}(t\theta)$ denotes the set of all variables in $t\theta$ and $\mathrm{dom}(\theta)$ denotes the domain of a substitution θ. In other words, the function returns the set of all program variables X such that V occurs in the instantiation of X under θ. The abstraction of a substitution θ in the sharing domain is defined as:

$$\alpha_{\mathrm{SH}}(\theta) = \{Occ(\theta, V) \mid V \in \mathrm{range}(\theta)\}$$

The concretization of an abstract substitution in the sharing domain is defined as

$$\gamma(\widehat{\theta}_{\mathrm{SH}}) = \{\theta \in Subst \mid \alpha_{\mathrm{SH}}(\theta) \subseteq \widehat{\theta}_{\mathrm{SH}}\}$$

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

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

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:

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

The component described above is essentially the abstract domain of Jacobs and Langen [15].

An abstract substitution in the freeness domain is a set of program variables (those that are known to be free). More formally, the abstraction and concretization functions in this domain are defined as follows:

$$
\begin{array}{lcl}
\alpha_{\mathrm{FR}}(\theta) & = & \{X \in \mathrm{dom}(\theta) \mid X\theta \in \mathit{Var}\} \\
\gamma(\widehat{\theta}_{\mathrm{FR}}) & = & \{\theta \in \mathit{Subst} \mid \alpha_{\mathrm{FR}}(\theta) \supseteq \widehat{\theta}_{\mathrm{FR}}\}
\end{array}
$$

The concretization of a sharing+freeness abstract substitution can be defined as the intersection of the concretizations of its two components:

$$\gamma(\widehat{\theta}) = \gamma(\widehat{\theta}_{\mathrm{SH}}) \cap \gamma(\widehat{\theta}_{\mathrm{FR}})$$

The set inclusion relation in the concrete domain induces a *partial order* on the abstract substitutions, i.e. $\widehat{\phi} \sqsubseteq \widehat{\psi}$ iff $\gamma(\widehat{\phi}) \subseteq \gamma(\widehat{\psi})$. The function *lub* computes the least upper bound of two abstract substitutions $\widehat{\phi}$ and $\widehat{\psi}$ by taking the least upper bound of their *sharing* and *freeness* components:

$$\mathit{lub}(\widehat{\phi}, \widehat{\psi}) = (\widehat{\phi}_{\mathrm{SH}} \cup \widehat{\psi}_{\mathrm{SH}}, \widehat{\phi}_{\mathrm{FR}} \cap \widehat{\psi}_{\mathrm{FR}})$$

It is important to point out that the approximations performed by the abstraction function and the *lub* function with respect to the sharing component imply that this component can actually represent in a compact way (rather than with an explicit disjunction) several combinations of sharing patterns. One of the main sources of information in being able to tell these combinations apart is the freeness information. In fact, sharing information is not independent of freeness information since known

freeness of certain variables restricts the allowable combinations of sharing patterns. 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.

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.

Variables in the freeness component are represented with dots, the rest with circles. The sharing patterns are represented with connected lines going to each variable of the corresponding sharing set. The resulting pictures are hypergraphs, i.e. graphs where the edges connect an arbitrary number of vertices.

Thus, the number of edges connected to a vertex is the number of sharing sets containing the corresponding variable, except for dot vertices (free variables) that can have multiple edges, all corresponding to the same sharing pattern, or none, meaning

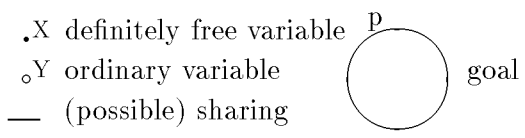


Figure 1: Types of objects in our pictorial representation.

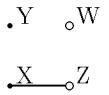
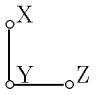
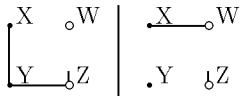
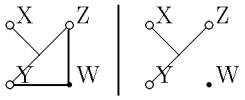
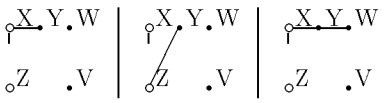
Variables	Abstract substitution	Representation
{X, Y, Z, W}	([[Y] [XZ]], [XY])	
{X, Y, Z}	([[XY] [YZ]], [])	
{X, Y, Z, W}	([[XYZ] [XW] [Y] [Z]], [XY])	
{X, Y, Z, W}	([[XYZ] [YZW] [W]], [W])	
{X, Y, Z, W, V}	([[X] [XY] [YZ] [W] [XYW] [V]], [YWV])	

Figure 2: Examples of representation of abstract substitutions

a sharing pattern with only this variable (since free variables must be in one and only one sharing pattern). Sharing sets that have no free variables are optional: they may or may not be active in a concrete substitution. A ground variable appears like an isolated circle.

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 is to the left and the second one to the right, and the variables present in both goals are put in the intersection.

Figure 2 shows several examples representing in one or more pictures abstract substitutions. The number of pictures corresponds to the number of alternative coverings of free variables by the sharing sets.

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

Before stating the conditions it is important to understand in which form a predicate can transform its abstract call substitution into its abstract answer substitution. Regarding the freeness component, what it can do is eliminate variables (instantiating

them). Regarding the sharing component, it can eliminate sharing sets (instantiating its variables to ground terms) or create more by union of the present sharing sets (unifying variables from these sharing sets). Note also that when a sharing set contains one or more free variables, if it is active, there is a single shared run-time variable corresponding to these program variables. Remember also that two sharing sets containing the same free variable cannot be active at the same time.

As mentioned in the introduction we will consider the parallelization of pairs of goals. Let $\tilde{p}$ and $\tilde{q}$ be two goals or sequences of goals, where $\tilde{q}$ is to the right of $\tilde{p}$. Also let $\widehat{\beta}$ and $\widehat{\psi}$ be the abstract call and answer substitutions for $\tilde{p}$. So the situation is $\{\widehat{\beta}\}\ \tilde{p}\ \{\widehat{\psi}\}\ \ldots\ \tilde{q}$. We define the sets:

$$\begin{aligned} \mathrm{S}(\tilde{p}) &= \{L \in \widehat{\beta}_{\mathrm{SH}} \mid L \cap \mathrm{var}(\tilde{p}) \neq \emptyset\} \\ \mathrm{SH} \ &= \mathrm{S}(\tilde{p}) \cap \mathrm{S}(\tilde{q}) = \{L \in \widehat{\beta}_{\mathrm{SH}} \mid L \cap \mathrm{var}(\tilde{p}) \neq \emptyset \ \wedge\ L \cap \mathrm{var}(\tilde{q}) \neq \emptyset\} \end{aligned}$$

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

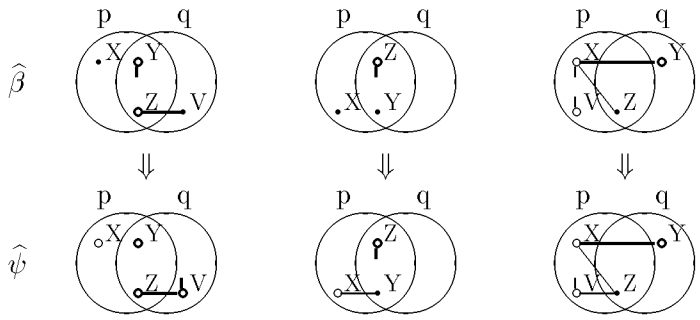
The following are our conditions for non-strict independence between $\tilde{p}$ and $\tilde{q}$:

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

Condition C1 deals with preserving freeness of shared variables*. By checking that all sharing sets of 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. Condition C2 is needed to preserve independence of shared variables: $N_1 ... N_k$ are sharing sets that $\tilde{p}$ can unite (thus they come from $\mathrm{S}(\tilde{p})$) 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 $\neg$C2 holds, $\tilde{p}$ can possibly bind the two independent shared variables.

*We would like to thank M. Bruynooghe for suggesting improvements to our original C1.

Three examples in which C1 fails: run-time shared variables can be further instantiated



Three examples in which C2 fails: run-time shared variables can aliase each other

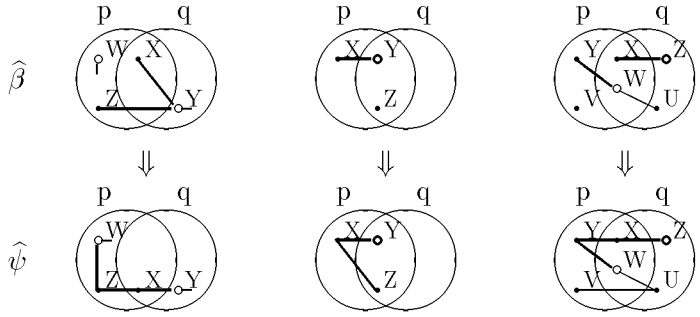


Figure 3: Situations where the conditions do not hold, and thus the goals are possibly not NSI

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 $\widehat{\psi}$.

4 Run-Time Checks for Non-Strict Independence

In the previous section 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.

In the previous section we proposed two conditions which had to hold for parallelization. Let us analyze what to do when each of the conditions is violated.

4.1 Condition C1 Violated

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

In this case we need run-time checks to ensure that the sharing sets $L \in \mathrm{SH}$ 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 overlaping, 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; they must be tried in this order:

- 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 $\mathrm{X} \in \mathrm{var}(\tilde{\mathrm{p}}) \cap \mathrm{var}(\tilde{\mathrm{q}})$ then we always are in this case, since all sharing sets containing X are in SH, so the ones that are legal contain free variables that remain free after executing $\tilde{\mathrm{p}}$, 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 $\mathrm{X} \in \mathrm{var}(\tilde{\mathrm{p}})$ and $\mathrm{Y} \in \mathrm{var}(\tilde{\mathrm{q}})$. Note that

the sharing sets in SH have a variable in both var($\tilde{p}$) and var($\tilde{q}$) or have one variable in var($\tilde{p}$) and another variable in var($\tilde{q}$). And, since the illegal sharing sets are in 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 SH, so they have free variables that remain free after executing $\tilde{p}$. 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 $\mathcal{F} = [\,]$.

4.2 Condition C2 Violated

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

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.

Now, for each L in the above formula, we compute the different groups of $N_1 ... N_k$ that $\tilde{p}$ can unite to give the sharing set L, without taking into account the number of sharing sets N_i that are in SH. The groups that have more than one sharing set in 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 $\tilde{p}$ 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), q(X,Y,W)`" and the abstract call and answer substitutions for "p(X,Y,Z)" are, respectively, $\widehat{\beta}$ = ([[X] [Y] [XY] [Z] [ZW]], [Y]) and $\widehat{\psi}$ = ([[X] [XY] [Z] [ZW]], [Y]). We have SH = [[X] [Y] [XY] [ZW]], and the illegal sharing sets for the first condition are [X] and [ZW]. The check for [X] is `allvars(X,[Y])`, since X can contain occurences of Y, given that [XY] is legal. The check for [ZW] is `ground(W)`, since there are no legal sharing sets containing W.

Althought condition C2 did not hold with the initial sharing sets, once ensuring that [X] is no longer active the condition is fulfilled, so we are ready to parallelize the two goals, the result being (here we omit the substitution of variables, to be explained in the next section):

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

where "`A -> B; C`" is the prolog if-then-else construction and "`&`" is the (unconditional) parallel operator.

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. [12] 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 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 a `indep/2` check on every pair).

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

5 Renaming and Substituting Variables

When using non-strict independence, and in order to prevent partial answers of a branch that ultimately fail from pruning the search space of other goals, parallel goals are in principle run in independent environments (see [12, 13]). The standard solution for this problem is a run-time transformation of the goals to be executed in parallel. This transformation involves eliminating any shared variable among parallel goals by renaming or substituting its occurrences so that no two occurrences in different goals remain the same, and adding some unification goals after the parallel conjunction to reestablish the lost links. This operation can be encoded at compile-time by performing `copy_term`'s of every goal and unifying the original goals and the copied versions after

the parallel conjunction. We will now propose more efficient methods which are based on the knowledge gathered during the annotation process. Note that a mere renaming of variables at compile-time is not sufficient in general: we can have terms with shared variables inside. So we use the following predicate:

`subst_vars(`$[X_1,\ldots,X_n]$`,`$[X'_1,\ldots,X'_n]$`,Z,Z')` :-
Z' *is a term equal to* `Z` *but with variables* $X'_1,\ldots,X'_n$ *in place of variables* $X_1,\ldots,X_n$*, respectively.*

We are interested in the potential run-time shared variables, but with the conditions and/or the checks we ensure that these are the free variables (those of $\widehat{\beta}_{\mathrm{FR}}$) that appear in the sharing sets of SH. So, the transformation procedure proceeds as follows:

- Group in sets the free variables that appear in the sharing sets of SH, so that those that appear in the same sharing set are grouped together, and the rest form sets with an unique element. This is so because if two free variables appear in the same sharing set, they are possibly aliased at run-time, so they need to be processed together.
- For each of those sets of free shared variables V:
  - compute $\mathrm{R}(V) = \{w \mid \exists\, L \in \mathrm{SH}\ \exists\, v \in V \quad v \in L \wedge w \in L \wedge w \notin V\}$, i.e. the set of the variables that appear in the sharing sets of SH with variables from V, excluding those of V. So they possibly contain at run-time variables from V.
  - Then, for each goal $\tilde{\mathrm{g}}$, the necessary renamings or substitutions regarding V are computed. Let $\mathcal{V} = \mathrm{var}(\tilde{\mathrm{g}}) \cap V$ and $\mathcal{R} = \mathrm{var}(\tilde{\mathrm{g}}) \cap \mathrm{R}(V)$. We will represent a renaming of a variable v as "ren(v)" and a substitution of a variable v inside w as "sv(v, w)". There are thre cases:
    - $\mathcal{V} = \emptyset, \mathcal{R} = \emptyset \rightarrow$ none.
    - $\mathcal{V} = \emptyset, \mathcal{R} \neq \emptyset \rightarrow \mathrm{sv}(v, w)$ for each $w \in \mathcal{R}$, where $v \in V$.
    - $\mathcal{V} \neq \emptyset \rightarrow \mathrm{ren}(v), \mathrm{sv}(v, w)$ for each $w \in (\mathcal{R} \cup \mathcal{V} - \{v\})$, where $v \in \mathcal{V}$.
  - Since for each V we need to transform all the goals minus one, the goal with the most expensive transformation is not considered. Substitutions are more expensive than renamings, substitutions in ordinary variables are more expensive than substitutions in free variables (which are in fact conditional unifications).

- Once computed the transformations for all the sets of variables, then for each goal the substitutions in the same variable are joined in a `subst_vars` predicate. Unification ("back-binding") goals must be included after the parallel conjunction for all the free variables renamed or substituted. Note that one side of these unifications is allways a free variable, since the conditions ensure that the first goal do not instantiate shared variables.

As an example, consider the parallel expresion `p(T,V,W)&q(U,V,W,X,Y)&r(W,Z)`, with the abstract call substitution $\widehat{\beta}$ = ([[T] [UV] [UVY] [VWX] [X] [XY] [Z]], [TUWY]). The sharing sets shared are SH = [[UV] [UVY] [VWX]]. We have two sets of free variables from SH: {U, Y} and {W}, with R({U, Y}) = {V} and R({W}) = {V, X}. The following table shows, for each of these sets, and for each goal, the values of $\mathcal{V}$ and $\mathcal{R}$ and the transformation needed.

	p(T,V,W)			q(U,V,W,X,Y)			r(W,Z)		
	$\mathcal{V}$	$\mathcal{R}$	*transformation*	$\mathcal{V}$	$\mathcal{R}$	*transformation*	$\mathcal{V}$	$\mathcal{R}$	*transf.*
{U, Y}	$\emptyset$	{V}	sv(U, V)	{U, Y}	{V}	ren(U), sv(U, Y), sv(U, V)	$\emptyset$	$\emptyset$	$\emptyset$
{W}	{W}	{V}	ren(W), sv(W, V)	{W}	{V, X}	ren(W), sv(W, V), sv(W, X)	{W}	$\emptyset$	ren(W)

In either rows we discard the transformation for the goal `q/5`. The two substitutions for the goal `p/3` are on the same variable, so they must be joined. Therefore, the parallel expresion is transformed into:

```
subst_vars([U,W],[U1,W1],V,V1),
p(T,V1,W1) & q(U,V,W,X,Y) & r(W2,Z),
U=U1, W=W1, W=W2
```

Figure 4 illustrates in pictures the transformation done, the bidirectional arrows showing the bindings performed by the back-binding goals. There are two situations depending on the covering of the free variables by the sharing sets.

6 Example Parallelization of a Program

As an example in this section we will show how to apply the proposed methods to a concrete program (quicksort using difference lists) in order to exploit the non-strictly independent and-parallelism it contains. Although the program is small, we think that

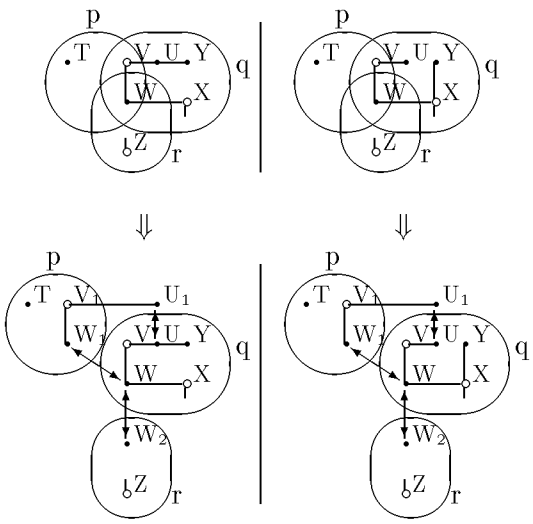


Figure 4: Representation of the effect of variable substitution in a parallel expresion.

it is of sufficient entity to show the potential of the proposal, and at the same time it is small enough to allow presenting the complete parallelization process.

The quicksort program we will use follows, with the abstract substitutions obtained by the analyzer annotated at each point of the program:

```
qsort(I,O) :-                              %[[O]],[O]
        qsort(I,O,[]).                     %[],[]
qsort([],L,L).
qsort([X|Xs],L,L2) :-                      %[[L],[L2],[Left],[Right],[L1]],
                                           %[L,Left,Right,L1]
        partition(Xs,X,Left,Right),        %[[L],[L2],[L1]],[L,L1]
        qsort(Left,L,[X|L1]),              %[[L,L1],[L2]],[L1]
        qsort(Right,L1,L2).                %[[L,L2,L1]],[]
partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right) :-      %[[Right],[Left1]],[Right,Left1]
        E=<C,                              %[[Right],[Left1]],[Right,Left1]
        !,
        partition(R,C,Left1,Right).        %[],[]
partition([E|R],C,Left,[E|Right1]) :-      %[[Left],[Right1]],[Left,Right1]
        E>C,                               %[[Left],[Right1]],[Left,Right1]
        partition(R,C,Left,Right1).        %[],[]
```

We will concentrate on the parallelization of the `qsort/3` predicate. Firstly, we will analyze whether it is possible to parallelize the first and second goal of the recursive clause of `qsort/3`, so we have that $\tilde{p}$ = `partition(Xs,X,Left,Right)` and $\tilde{q}$ = `qsort(Left,L,[X|L1])`, and the abstract substitutions involved are $\widehat{\beta}$ = ([[L] [L2] [Left]

$[Right]\,[L1]], [L\ Left\ Right\ L1])$ and $\widehat{\psi} = ([[\mathrm{L}]\,[\mathrm{L2}]\,[\mathrm{L1}]], [\mathrm{L\ L1}])$. Then, we compute the set SH = [[Left]]. Condition C1 is not met, since "Left" is not in $\widehat{\psi}_{\mathrm{FR}}$, and furthermore this is a free variable that does not appear in another sharing set in $\widehat{\beta}_{\mathrm{SH}}$, so it is sure that the goals are not non-strictly independent. In a similar manner it can be shown that the first and third goal of the clause are not non-strictly independent either.

Finally, let us try with the second and third goals in the same clause. Now $\tilde{\mathrm{p}}$ = `qsort(Left,L,[X|L1])`, $\tilde{\mathrm{q}}$ = `qsort(Right,L1,L2)`, $\widehat{\beta} = ([[\mathrm{L}]\,[\mathrm{L2}]\,[\mathrm{L1}]], [\mathrm{L\ L1}])$ and $\widehat{\psi} = ([[\mathrm{L\ L1}]\,[\mathrm{L2}]], [\mathrm{L1}])$. The shared sharing sets are SH = [[L1]]. But now the conditions hold: L1 $\in \widehat{\psi}_{\mathrm{FR}}$ and no sharing sets meet $\neg$C2. Thus in this case we have non-strict independence, and no run-time checks are needed (note also that the goals are not strictly independent, since they share the free variable "L1").

The last step is to see whether we need to rename or substitute any variable in the goals. In both goals we only need to rename the variable "L1", so the predicate `qsort/3` would be left as:

```
qsort([],L,L).
qsort([X|Xs],L,L2):-
        partition(Xs,X,Left,Right),
        qsort(Left,L,[X|L1]) & qsort(Right,L1_prime,L2),
        L1=L1_prime.
```

7 Some experimental results

We have measured the speedups of five programs that have non-strict independence but have no strict independence, relative to the execution on one processor. These programs have been parallelized using the techniques presented so far. The results are given in table 1. The "Max" column shows the maximum speedup found and the number of processors needed (in fact, we stopped when the increment of speedup was less than 5%). We believe that the results obtained are quite encouraging.

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 128 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 89 elements with recursive "depth" of five. 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 eight rings. The **qsort** program is the one shown in the previous section. The speedups were measured sorting a list

Bench	# of processors									Max
	2	3	4	5	6	7	8	9	10	
array2list	1.94	2.80	3.59	4.33	5.01	5.65	6.24	6.75	7.24	12.27 (29)
flatten	1.98	2.90	3.77	4.54	5.27	5.94	6.57	7.14	7.67	13.87 (32)
hanoi_dl	1.99	2.94	3.86	4.75	5.60	6.43	7.23	7.99	8.74	24.30 (50)
qsort	1.83	2.40	2.75	3.02	3.18	3.32	3.42	3.49	3.54	3.54 (10)
sparse	1.90	2.63	3.32	3.62	4.35	4.54	5.23	5.33	5.44	6.34 (15)

Table 1: Speedups of several programs with NSI

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 16×16 elements, with 12 positive elements.

8 Beyond Sharing+Freeness Analysis

In the previous sections we have assumed that we only had the information from sharing+freeness analysis. In this section we briefly discuss what can be done when more information is available.

8.1 Information about "Purity" of Predicates

If we examine the conditions for parallelization stated in previous sections, we can see that only the behavior of the first goal $\tilde{p}$ is considered. However, if $\tilde{q}$ has certain properties, for example, if $\tilde{q}$ is known to be "pure" in a certain sense (i.e. it has no extra-logical builtins which are sensitive to variable instantiation), the conditions can be further relaxed [14]. The resulting *generalized* definition of non-strict independence is:

> Consider a collection of goals $g_1, \ldots, g_n$ and a given substitution θ. Consider also the set of shared variables $SH = \{v \mid \exists i, j, 1 \leq i < j \leq n, v \in (var(g_i\theta) \cap var(g_j\theta))\}$ and the set of goals containing each shared variable $G(v) = \{g_i\theta \mid v \in var(g_i\theta), v \in SH\}$. Let θ_i be any answer substitution for $g_i\theta$. The given collection of goals is non-strictly independent for θ if the following conditions are satisfied:

- $\forall x, y \in SH$, $\exists$ at most one $g_i\theta$ such that for any θ_i we have that $\{x, y\}\theta_i \not\equiv \{x, y\}$;
- $\forall x, y \in SH$, if $\exists\ g_i\theta$ meeting the condition above, then $\forall g_j\theta, j > i$, such that $\{x, y\} \cap var(g_j\theta) \neq \emptyset$, g_j is a pure goal, and $\{x, y\}\theta_j \equiv \{x, y\}$ for all θ_j which are partial answers during the execution of $g_j\theta$.

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

Intuitively, the first condition of the above definition requires that at most one goal modify a shared variable or alias a pair of variables. The second condition does not require that it be the rightmost goal containing the variables, but it does require that any goals to the right of the one modifying the variables be pure and do not "touch" such variables. This ensures that its search space could not have been pruned by any bindings made to those variables and therefore it is safe to run it in parallel, i.e. isolated from such bindings by the renaming transformation.

In practice, determining whether goals are pure or not is quite easy, and can also be used for other purposes (for example, in the &-Prolog compiler such an analysis is performed anyway for side-effect sequencing).

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

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

Condition C1′ differs from C1 in that it allows p̃ to further instantiate a shared variable, provided that this variable is not touched by q̃ (q̃ 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 q̃ (note that only if q̃ further instantiates the two variables can it possibly be affected by these bindings).

Related analyses, like purity of variables in predicates, would provide better information for the parallelization, but the definition of this type of information needs to

be clarified and is beyond the scope of this paper.

8.2 Towards an Improved Analysis for Non-Strict Independence

Although, in general, a more precise analysis is not always necessarily a better analysis (because more accurate information requires more time), it is certain that for different purposes we want different pieces of information and that the analysis used so far can be improved.

In the case of our study, we think that the key idea is to have access to the greatest number of run-time free variables, in order to check their possible instantiations, having at the same time more accurate information regarding sharing. To achieve this goal, sharing and freeness could be combined with other analyses, like linearity [24], depth-k [22], or with a recursive type analysis, mainly for lists, to deal, for example, with lists of free variables (see [2, 17, 1]). All these alternatives will be taken into account in further work. However, note that the approach presented is still valid directly or with slight modifications for these more sophisticated types of analyses. For example, if a depth-k analysis is used in combination with sharing+freeness, the information in the sharing+freeness will refer to the variables in the depth-k terms, and the same parallelization conditions would apply.

9 Conclusions

Despite the advantage of "non-strict" independence (NSI) over "strict" independence (SI) in terms of generality and the amount of parallelism it can exploit, all compilation technology developed to date has been based on SI, due to the complexity of compile-time detection of NSI. In an attempt to fill this gap we have presented several techniques for achieving this compile-time detection. 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. We have presented an efficient algorithm for performing combined compile-time/run-time renaming of variables to ensure that non-strictly independent goals run in separate environments with respect to their shared variables. An example of the application of the proposed methods to a concrete program has also been given, along with some experimental results showing the speedups found in this and other

programs presenting non-strict independence. Finally, we have also treated the case where additional information about purity of predicates is available, giving techniques for detection of NSI in the context of this information.

We have implemented these algorithms and are currently interfacing them with the sharing+freeness analyzer implementation available to us with the objective of obtaining a complete compile-time parallelizer capable of detecting NSI. We are also planning on looking, in the light of the techniques developed, at other types of analyses which may provide additional information useful for such parallelization.

References

1. A. Bansal and L. Sterling. An Abstract Interpretation Scheme for Identifying Inherent Parallelism in Logic Programs. *New Generation Computing*, (7):273–324, 1990.

2. M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *Fifth International Conference and Symposium on Logic Programming*, pages 669–683, Seattle, Washington, August 1988. MIT Press.

3. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. Technical Report TR Number CLIP7/93.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, October 1993.

4. Michael Codish, Dennis Dams, Gilberto File, and Maurice Bruynooghe. Freeness Analysis for Logic Programs - And Correctness? In *Proc. Int'l. Conf. on Logic Programming*. MIT Press, 1993. To appear.

5. 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.

6. A. Cortesi and G. File. Abstract Interpretation of Logic Programs: an Abstract Domain for Groundness, Sharing, Freeness and Compoundness Analysis. In *ACM Symposium on Partial Evaluation and Semantic Based Program Manipulation*, pages 52–61, New York, 1991.

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

8. V. Dumortier, G. Janssens, M. Bruynooghe, and M. Codish. Freeness Analysis in the Presence of Numerical Constraints. In *Tenth International Conference on Logic Programming*, pages 100–115. MIT Press, June 1993.

9. G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.

10. S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46. MIT Press, June 1990.

11. M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

12. M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

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

14. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 1993. To appear.

15. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

16. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

17. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

18. K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

19. K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

20. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

21. B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 313–331. MIT Press, October 1989.

22. T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

23. K. Shen. *Studies in And/Or Parallelism in Prolog.* PhD thesis, U. of Cambridge, 1992.

24. H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

25. R. Sundarajan. An Abstract Interpretation Scheme for Groundness, Freeness, and Sharing Analysis of Logic Programs. Technical Report CIS-TR-91-06, U. of Oregon, Eugene, Oregon 97403, October 1991.

26. D.H.D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

27. H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.