Theoretical Computer Science 410 (2009) 4704–4723

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Non-strict independence-based program parallelization using sharing and freeness information

Daniel Cabeza Gras [a], Manuel V. Hermenegildo [a,b,*]

[a] Facultad de Informática, Universidad Politécnica de Madrid (UPM), Spain

[b] IMDEA Software – Madrid Institute for Advanced Studies in Software Development Technology, Spain

ARTICLE INFO

Keywords:
Parallelism
Automatic parallelization
Abstract interpretation
Abstract domains
Sharing and freeness
Non-strict independence
Parallelizing compilers
Declarative languages
Logic programming

ABSTRACT

The current ubiquity of multi-core processors has brought renewed interest in program parallelization. Logic programs allow studying the parallelization of programs with complex, dynamic data structures with (declarative) pointers in a comparatively simple semantic setting. In this context, automatic parallelizers which exploit and-parallelism rely on notions of independence in order to ensure certain efficiency properties. "Non-strict" independence is a more relaxed notion than the traditional notion of "strict" independence which still ensures the relevant efficiency properties and can allow considerable more parallelism. Non-strict independence cannot be determined solely at run-time ("a priori") and thus global analysis is a requirement. However, extracting non-strict independence information from available analyses and domains is non-trivial. This paper provides on one hand an extended presentation of our classic techniques for compile-time detection of non-strict independence based on extracting information from (abstract interpretation-based) analyses using the now well understood and popular *Sharing* + *Freeness* domain. This includes algorithms for combined compile-time/run-time detection which involve special run-time checks for this type of parallelism. In addition, we propose herein novel annotation (parallelization) algorithms, URLP and CRLP, which are specially suited to non-strict independence. We also propose new ways of using the *Sharing* + *Freeness* information to optimize how the run-time environments of goals are kept apart during parallel execution. Finally, we also describe the implementation of these techniques in our parallelizing compiler and recall some early performance results. We provide as well an extended description of our pictorial representation of sharing and freeness information.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

The use of multi-core processors is becoming widespread and this is bringing parallel computing clearly into the mainstream. As an example, most laptops currently on the market contain two cores (capable of running up to four threads simultaneously) and single-chip, 8-core servers are also in widespread use. Furthermore, the trend is that the number of on-chip cores will double with each processor generation. In this context, being able to exploit such parallel execution capabilities in programs as easily as possible becomes more and more a necessity. However, it is well-known [1,42] that parallelizing programs is a hard challenge. This has brought renewed interest in language-related designs and tools which can simplify the task of producing parallel programs, and, in particular, in parallelizing compilers and declarative languages.

The interest in declarative programming paradigms is due to the fact that their high-level of abstraction brings significant advantages to the parallelization task. Logic programs are particularly interesting because on one hand they exhibit these

* Corresponding author at: Facultad de Informática, Universidad Politécnica de Madrid (UPM), Spain.
E-mail address: herme@fi.upm.es (M.V. Hermenegildo).

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.07.044

favorable characteristics, stemming from their high level and declarative nature, and at the same time they pose, in a semantically clean and well-understood setting, challenges which relate closely to the most difficult scenarios faced in traditional parallelization [27]. In particular, interesting challenges faced during the parallelization of logic programs include the presence of dynamically allocated, complex data structures containing "(declarative) pointers" (logical variables), non-trivial notions of independence, the presence of highly irregular computations and dynamic control flow, and having to deal with speculative computations and search. As a result, advances in the parallelization of logic programs also shed light on the parallelization of current and future imperative languages.

Logic programs exhibit several kinds of parallelism [13,24], among which or- and and-parallelism are the most exploited in practice. In this paper we concentrate on and-parallelism, which manifests itself in applications in which a given problem can be divided into a number of independent sub-problems. For example, it appears in algorithms where independent, possibly recursive calls or loop iterations can be executed in parallel (simple, well-known examples are quick-sort or matrix multiplication). Some examples of systems which exploit and-parallelism are &-Prolog [10,26,29] (and, more recently, its successor Ciao [28,30]), ROPM [56], AO-WAM [23], ACE [54,55], DDAS/Prometheus [58,59], systems based on the "Extended" Andorra Model [63] such as AKL [41], etc. (please see their references and [24] for other related systems).

The objective of the parallelization process performed by a parallelizing compiler is to uncover as much as possible of the available parallelism in the program, while guaranteeing that the correct results are computed –i.e., *correctness*– and that other observable characteristics of the program, such as execution time, are improved (speedup) or, at the minimum, preserved (no-slowdown) —i.e., *efficiency*. A central issue is, then, under which conditions two parts of a (logic) program can be *correctly* and *efficiently* parallelized. All of the systems exploiting and-parallelism mentioned above rely on some notion of independence (also referred to as "stability" [25]) among non-deterministic goals being run in and-parallel in order to ensure these important efficiency properties.[1] Two basic notions of independence for logic programs are strict and non-strict independence [33–35]. Other more general notions have been developed based directly on search space preservation and which are applicable to constraint logic programs [17,18], but herein we concentrate on the former classic notions for the Herbrand domain.

1.1. Strict independence

Strict independence corresponds to the traditional notion of independence, normally applied to goals [13,21,29]: two goals g_1 and g_2 are said to be strictly independent for a substitution θ iff $var(g_1\theta) \cap var(g_2\theta) = \emptyset$, where $var(g)$ 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 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 slow-down condition can be ensured [33–35].[2]

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, i.e., calls to `ground/1` (which tests for groundness of its argument) or `indep/2` (which succeeds if its two arguments share no variables).

As an example, consider the clause fragment "`p(X,Y), q(Y,Z)`". The run-time tests that ensure the strict independence of `p(X,Y)` and `q(Y,Z)` are `ground(X)` and `indep(Y,Z)`. Thus, transforming the fragment above into:

```
( ground(X), indep(X,Y) -> p(X,Y) & q(Y,Z)
                        ;  p(X,Y) , q(Y,Z) )
```

(where "`A -> B; C`" is the Prolog *if-then-else* and "`&`" is the &-Prolog parallel conjunction operator) ensures that `p/2` and `q/2` will run in parallel if they are strictly independent and sequentially otherwise.

1.2. The parallelization process

A considerable body of work exists on the task of automatically parallelizing programs at compile time using strict (as well as other types of) independence [8,21,48,50]. A detailed overview of this work is beyond the scope of this paper —see [24,27] for tutorial introductions and additional pointers to literature. We do however provide a brief introduction to the process, based on the methodology used in the &-Prolog/Ciao parallelizer,[3] originally proposed in [36] (see Fig. 1 representing the parallelization of three generic literals "`g1(...), g2(...), g3(...)`").

[1] It has been noted [27,28] that independent and dependent and-parallelism are simply the application of the same principle, *independence*, at different levels of granularity in the computation model.

[2] Note that, unlike in functional or imperative programs, the risk involved in running an arbitrary goal in parallel in a logic program is not only that it may lead to a run-time error or a wrong answer. More subtly, the program may finish without error and provide the right answer, but only after the exploration of much larger search spaces (sometimes infinite) than those corresponding to the sequential execution, and, thus, at the cost of significant slow-downs.

[3] This parallelizer is part of the Ciao preprocessor, CiaoPP. This system is a state-of-the-art interactive programming environment, which offers abstract modular interpretation-based analysis, debugging, verification, and optimization —see [30] for a description of the system with pointers to literature.

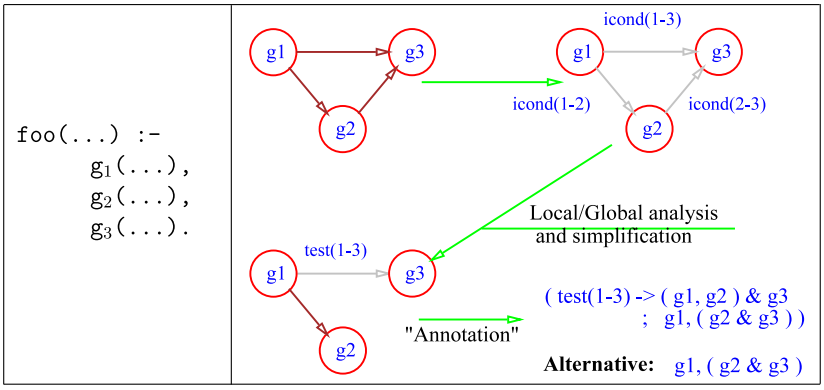


Fig. 1. Parallelizing "$g_1(\ldots)$, $g_2(\ldots)$, $g_3(\ldots)$" in &-Prolog/CiaoPP.

Typically, a dependency graph is first built which in principle reflects the total ordering of statements and calls given by the sequential semantics. Traditionally, for strict independence each edge in the graph is then labeled with the run-time data conditions (the run-time check) that would guarantee independence of the statements joined by the edge. As mentioned above, in the case of strict independence these conditions are groundness of certain variables or absence of shared variables among two variables each belonging to one of the two goals being considered (corresponding to the calls to the `ground/1` or `indep/2` run-time tests).

If the appropriate option is selected, the parallelizer then obtains information about the possible run-time substitutions (e.g., sharing and freeness information) at all points in the program as well as other types of information from a *Global Analyzer*, using techniques [4,5,39,40,49,52] that are generally based on abstract interpretation [16].[4] This information is then used to prove the conditions in the graph statically to be true or false. If a condition is proved to be true, then the corresponding edge in the dependency graph is eliminated. If proved false, then an unconditional edge (i.e., a static dependency) is left. Still, in other edges conditions may remain, but possibly simplified.

In the next step the annotator then encodes the resulting graph, producing an "annotated" (parallelized) program. The techniques used for performing this process depend on many factors including whether arbitrary parallelism or just fork-join structures are allowed[5] and also whether run-time independence tests are allowed or not. As an example, Fig. 1 presents two possible encodings in &-Prolog of the (schematic) dependency graph obtained after analysis. The parallel expressions generated in this case use only fork-join structures, one with run-time checks and the other one without them.

1.3. Non-strict independence

Non-strict independence is a relaxation of strict independence, defined (for goals) as follows [6,34,35]:

> 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 $g_i\theta$, i.e., θ_i is any of the final substitutions obtained from evaluating $g_i\theta$ using SLD-resolution until an empty resolvent is obtained. We assume substitutions to be represented using equalities (in CLP style), i.e., a binding of two variables x and y is represented as $x = y$. Thus, we consider the bindings x/y and y/x equivalent. We use $\dot{\exists}$ as usual to represent "there exists at most one". The given collection of goals is non-strictly independent for θ if the following conditions are satisfied:
> - $\forall x, y \in SH$, $\dot{\exists}\, g_i\theta$ such that for some answer substitution θ_i for $g_i\theta$ we have that $x\theta_i$ is not a variable 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 $g_j\theta$ $x\theta_j$ is a variable and $x\theta_j \neq y\theta_j$ if $x \neq y$.

Intuitively, the first condition of the above definition requires that at most one goal further instantiates a shared variable or aliases a pair of variables. The second condition requires that any goal to the right of the one modifying the variables be pure and not "touch" such variables during its execution. This ensures that its search space could not have been pruned by any bindings made to these variables and therefore it is safe to run it in parallel.

Here pure is applied to a goal that has no extra-logical built-ins which are sensitive to variable instantiation. For example, if one of the goals to the right of the one modifying the variables contains a call to the Prolog built-in `var/1` (which only succeeds if a variable is *free*) its search space could be reduced by a binding produced to its left (i.e., earlier, in sequential

[4] The parallelizer also receives information from the *Side-Effect Analyzer* on whether or not each non-built-in predicate and clause of the given program is *pure*, or contains a *side-effect*, and dependencies are added to correctly sequence such side-effects. The parallelizer also receives optionally information on goal granularity from cost analysis (see [19,20,30,46] and its references) which is used, e.g., to avoid parallelizations when tasks are too small.

[5] For an example where expressions do not need to be fork-join see [8,9].

execution), and this reduction might be lost during parallel execution. Similarly, if one of the goals to the right contains a call to the Prolog built-in ==/2 (which only succeeds if two variables are *identical*), its search space could be reduced by a variable to variable binding produced to its left, which might again be lost during parallel execution. Also some side effects, such as printing the value of a variable, are obviously sensitive to bindings.[6]

The definition above is a generalization of that originally given in [34]. In the case in which no information is available on the purity of goals (and, thus, all goals have to be conservatively assumed to be impure) the definition of non-strict independence does not need to be based on partial answers and 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 $x, y \in var(g_i\theta)$ has an answer substitution θ_i for which $x\theta_i$ is not a variable 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 the variable, 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.

Clearly, non-strict independence is a more powerful notion than strict independence, since strictly independent goals are always non-strictly independent. The crucial point is that non-strictly independent goals preserve the same properties as strictly independent ones with respect to correctness and efficiency in parallel execution [34,35]. In addition, studies aimed at detecting the intrinsic, ideal amounts of parallelism present in logic programs already pointed out that there is potential to obtain significant additional speedups from the exploitation of non-strict independence [59]. In practice, non-strict independence has wide application for example in the parallelization of programs which use difference lists, and incomplete structures in general. An example of this is the following `flatten` program, which eliminates nestings in lists (the Ciao "pred" assertion [30] states that the mode considered implies taking calling `flatten/2` with a ground term in the first argument and a variable in the second):

```
:- pred flatten/2 : ground * var.

flatten(Xs,Ys) :- flatten(Xs,Ys,[]).

flatten([], Xs, Xs).
flatten([X|Xs], Ys,   Zs) :- flatten(X,Ys,Ys1), flatten(Xs,Ys1,Zs).
flatten(  X,  [X|Xs], Xs) :- atomic(X), X \== []
```

When run in this "forwards" mode this program unnests a list without copying by creating open-ended lists and passing a pointer to the end of the list (`Ys1`) to the recursive call. Since this variable is not bound by the first call to `flatten/3` in the body of the recursive clause, the calls to `flatten(X,Ys,Ys1)` and `flatten(Xs,Ys1,Zs)` are (non-strictly) independent and all the recursions can be run in parallel. Note that `Ys1` may be aliased to `Ys` by the first goal, but `Ys` is known to be free and not shared with the second goal.

Exploiting non-strict independence is non-trivial due to at least two factors. The first one is that, as mentioned before, 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. In addition, 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 inferred by global analyses and some non-trivial translation is required.

1.4. Towards non-strict independence through sharing and freeness

A number of early studies [33–35] suggested that coupling sharing and groundness analysis with freeness analysis could be instrumental in the detection of both strict- and non-strict independence. This has been one of the motivations behind the development of analyzers capable of inferring these three types of information [2,12,14,37,38,44,45,51,53,61,64].

The task of providing the bridge between the availability of sharing and freeness information and actually being able to reason about the non-strict independence of a set of literals was first addressed in [6], which developed concrete techniques for determining non-strict independence at compile-time. This paper is on one hand an extended and improved version of [6]. More concretely, we expand [6] by providing extended explanations and a number of corrections, more and updated examples, updated references, a description of a complete run through the parallelizer including the results from

[6] In addition, many side effects (and in particular I/O) may need to be serialized, to preserve the order of observables, but that is a separate matter that is addressed by parallelizers in an orthogonal way to the application of the notion of independence.

the analyzer for the quick-sort program using difference lists, a more thorough description of the implementation and the parallelization process (all of it now integrated within the Ciao/CiaoPP system [30]), and a fuller description of the benchmarks used.

In addition, we propose URLP and CRLP, novel annotation (parallelization) algorithms specially suited to non-strict independence. We also propose novel techniques for using the *Sharing+Freeness* information to optimize how the run-time environments of goals are kept apart during parallel execution. These techniques are quite important for reducing run-time overhead and thus obtaining speedups. It is interesting to note that, while much progress has been made since [6] in the analyzers capable of inferring sharing and freeness information, to the best of our knowledge this work (and its implementation in the context of the Ciao/CiaoPP system) still represents the state of the art in terms of converting such analysis information into exploitable non-strict independent and-parallelism.

Following [6], we focus on a particular way of expressing sharing and freeness information: the *Sharing+Freeness* domain [51]. Concentrating on a particular domain allows a significant 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.

The rest of the paper proceeds as follows: Section 2 explains the *Sharing+Freeness* domain (which, as mentioned before, is the particular abstract interpretation domain for which the conditions for parallelism are given), and provides an extended description of our pictorial representation for the abstract substitutions involved. Section 3 presents the sufficient conditions for compile-time detection of non-strict independence. Section 4 deals with the combination of compile-time analyses and run-time checks for detecting non-strict independence, presenting the type of run-time checks required for this type of parallelism. It also connects this method with the techniques used for the detection of strict independence and shows that our techniques actually also provide improved run-time checks for strict independence. Section 5 proposes URLP and CRLP, the new annotation (parallelization) algorithms specially suited to non-strict independence. Section 6 then proposes techniques for using the *Sharing+Freeness* information to rename and replace variables in order to optimize how the run-time environments of goals are kept apart during parallel execution. Section 7 illustrates the techniques proposed by using them to parallelize a sample program. Section 8 recalls some experimental results showing the speedups obtained in several programs presenting non-strict independence but no strict independence. Finally, Section 9 provides conclusions and suggests future work.

2. Understanding *Sharing+Freeness* abstract substitutions

The *Sharing+Freeness* abstract domain [51] (other related analyses for which our results may be valid include the previously mentioned [2,12,14,37,38,44,45,51,53,61,64]) was proposed with the objective of obtaining at compile-time, by means of an abstract interpretation-based [16] analyzer, 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 that do not have 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 [39,40,49,52]; 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). However, for conciseness we will drop the commas that separate the elements in the abstract substitutions. 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:

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

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:

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

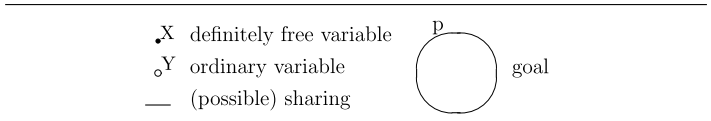


Fig. 2. Types of objects in our pictorial representation.

The component described above is essentially the abstract domain of Jacobs and Langen [39,40], for which Muthukumar et al. proposed more efficient abstract unification algorithms [40,49,52],[7] and which has recently received much attention [2,12,14,37,38,44,61], specially regarding the development of widenings in order to trade precision for performance in unfavorable cases [45,53,64].

An abstract substitution in the freeness domain is a set of program variables (those that are known to be free). The concretization of a *Sharing+Freeness* abstract substitution can be defined as the intersection of the concretizations of its two components.

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:

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

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 sets. The possible combinations of sharing sets represented by a *Sharing+Freeness* abstract substitution $\widehat{\theta}$ 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 ($\forall v \in \widehat{\theta}_{\mathrm{FR}} \exists L \in S\, v \in L$).

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 pictorial representation presented in the following section these issues will be greatly clarified.

2.1. Pictorial representation of substitutions

We use a pictorial representation of substitutions in the *Sharing+Freeness* domain [6] in order to make it easier to understand such abstract substitutions and also 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. Fig. 2 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.

As the *Sharing+Freeness* abstract substitutions give information in terms of program variables, only these variables appear in the figures.[8] 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.

[7] And also an improved precision of the abstract operations, based on considering the concrete information in clause heads and goals, although this can also be considered an optimization outside the abstract domain.

[8] Note, however, that if, as suggested in Section 9, the *Sharing+Freeness* domain is combined with another domain that introduces new variables (such as, for example, a depth-k domain [43,57]) then these variables would appear in the *Sharing+Freeness* component, and, thus, also in the pictures.

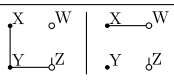
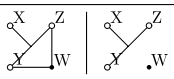
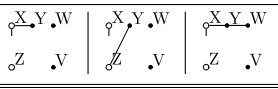
Variables / Abstract substitution	Pictorial Representation
{X, Y, Z, W} / ([[Y] [XZ]], [XY])	
{X, Y, Z} / ([[XY] [YZ]], [])	
{X, Y, Z, W} / ([[XYZ] [XW] [Y] [Z]], [XY])	
{X, Y, Z, W} / ([[XYZ] [YZW] [W]], [W])	
{X, Y, Z, W, V} / ([[X] [XY] [YZ] [W] [XYW] [V]], [YWV])	

Fig. 3. Examples of representation of abstract substitutions.

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".

Fig. 3 shows several examples representing different abstract substitutions through the corresponding picture(s). The number of pictures corresponds to the number of alternative coverings of free variables by the sharing sets.

- In the first example, the abstract substitution is represented with only one picture, as there is only one possible covering of the free variables by the sharing sets. W is ground, X and Y are free, and Z is a term that contains X.
- The second example, since there are no free variables, also represents an abstract substitution with only one picture. The two sharing sets represented are therefore optional, so in fact we have four possibilities depending on which of the two are active.
- In the third example we have two pictures: either the sharing set [XYZ] or the two sharing sets [XW] and [Y] are active, since these are the two possible coverings of the free variables X and Y. In the first picture W is ground, whereas Z is not ground and contains the variables X and Y, that are aliased, and may contain more variables. In the second picture, W is a term that contains X, Y is free and independent from the other variables, Z is also independent from the other variables (and it may be ground since its sharing set is optional).
- The fourth example shows two pictures, depending on whether [YZW] or [W] is active. In both pictures there is an optional sharing among X, Y and Z. In the first picture, Y and Z also share the variable W.
- Finally, the fifth example shows three pictures. In the first, the covering of the free variables Y, W and V is performed by the sharing sets [XY], [W] and [V]; in the second, by [X], [YZ], [W] and [V]; and in the third, by [XYW] and [V]. There are no more combinations that cover all the free variables without overlapping. In the first picture, V and W are free variables and independent, Z is ground, and X contains the free variable Y and possibly others. In the second picture, V and W are also free variables and independent, Z contains the free variable Y, and X is also independent (as in one of the previous cases, we do not know if it is free, ground, or otherwise). In the third picture, V is free and independent, Z is ground, and X contains Y and W, which are aliased, as well as possibly more variables.

3. Conditions for non-strict independence based on *Sharing+Freeness* analysis results

We now turn to our main objective of describing how conditions for non-strict independence can be derived from the information present in *Sharing+Freeness* abstract states obtained from global analysis. In the following we will focus on the case of analyzing the (non-strict) independence of *two* literals. This is convenient from a practical point of view because, as mentioned before, many parallelization algorithms work by repeatedly considering whether two literals are independent. 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 is fortunately based on sound theoretical foundations. 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 $SH = var(g_1\theta) \cap var(g_2\theta)$. 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 original definition, and thus considering only pairs of goals can be done without loss of generality.

Now, in order to derive conditions for detecting non-strict independence with respect to the information present in *Sharing+Freeness* abstract states, note that our definition of non-strict independence 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.

Before stating the conditions it is important to understand in which form a goal can transform its abstract call substitution into its abstract answer substitution:

- Regarding the freeness component, what it can do is eliminate variables from the component (by instantiating them).[9]
- Regarding the sharing component, it can eliminate sharing sets (by instantiating its variables to ground terms) or create more by union of the present sharing sets (by unifying variables from these sharing sets). If a variable in a sharing set is further instantiated but not made ground, the sharing set remains unchanged. 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. Recall also that two sharing sets containing the same free variable cannot be active at the same time.

The following two sections deal with the cases that arise depending on whether purity information is considered or not.

3.1. Conditions disregarding purity of goals

As mentioned before, we will consider the parallelization of pairs of program literals. First, we will state the conditions without purity information.

Let p and q be two literals, where q is to the right of p. Also let $\widehat{\beta}$ and $\widehat{\psi}$ be the abstract call and answer substitutions for p, i.e., the situation is $\{\widehat{\beta}\}$ p $\{\widehat{\psi}\}$ ... q. We define the sets:

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

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 active, contain run-time shared variables).

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

$$\mathrm{C1} \quad \forall L \in \widehat{\mathrm{SH}} \;\; L \cap \widehat{\psi}_{\mathrm{FR}} \neq \emptyset$$
$$\mathrm{C2} \quad \neg\,(\; \exists \{N_1, N_2, \ldots, N_k\} \subseteq \mathrm{S(p)},\ \text{with } card(X) > 1,\ \text{such that}$$
$$\forall i, j \;\; 1 \leq i < j \leq k \;\Rightarrow\; N_i \cap N_j \cap \widehat{\beta}_{\mathrm{FR}} = \emptyset$$
$$\wedge\, \exists L \in \widehat{\psi}_{\mathrm{SH}} \;\; \left(L = \bigcup_{i=1}^{k} N_i\right) \wedge N_1, N_2 \in \widehat{\mathrm{SH}} \;\;)$$

Condition C1 deals with preserving freeness of shared variables. By checking that all sharing sets of $\widehat{\mathrm{SH}}$ have a free variable in the abstract answer substitution $\widehat{\psi}$, we ensure 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 others 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: $N_1 \ldots N_k$ are sharing sets that p can join (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.

Fig. 4 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 $\widehat{\psi}$.

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.

[9] Note that this is true for pure goals. However, if a goal contains, e.g., a call to `var/1` it can actually *add* to the freeness component: the goal could have been called with a variable that is not known to the analysis to be definitely free, but, due to the call to `var/1`, the analysis may be able to determine that only the calls in which the variable is free can succeed and will leave it free.

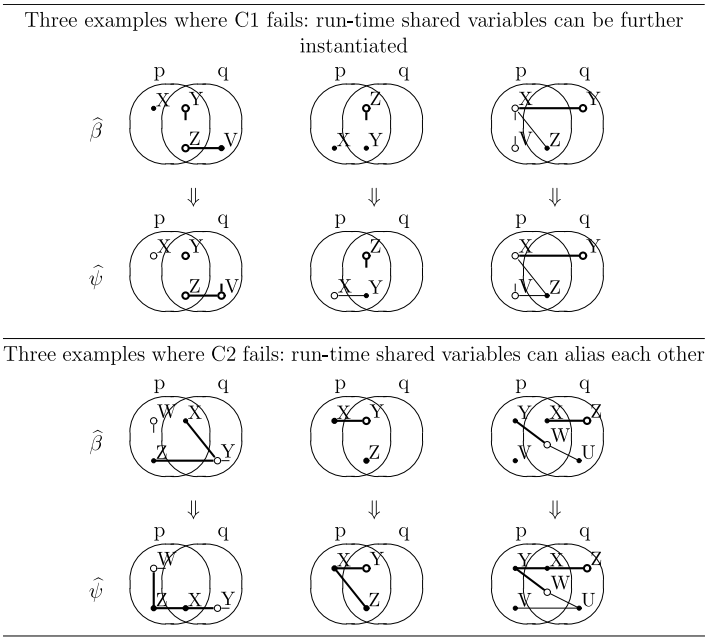


Fig. 4. Situations where the conditions do not hold, and thus the goals are possibly not non-strictly independent.

If we examine the conditions stated in the previous section, we can see that only the behavior of the first literal 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:

$$
\begin{array}{ll}
\text{C1}' & \forall\, L \in \widehat{\text{SH}}\ \ L \cap \widehat{\psi}_{\text{FR}} \neq \emptyset \ \vee\ L \cap \widehat{\phi}_{\text{FR}} \neq \emptyset \\
\text{C2}' & \neg\,(\ \exists\, \{N_1, N_2, \ldots, N_k\} \subseteq \text{S(p)}\ \text{ with }\ \mathit{card}(X) > 1,\ \text{such that} \\
 & \qquad \forall\, i,j\ 1 \leq i < j \leq k\ \Rightarrow\ N_i \cap N_j \cap \widehat{\beta}_{\text{FR}} = \emptyset \\
 & \qquad \wedge\, N_1 \cap \widehat{\phi}_{\text{FR}} = N_2 \cap \widehat{\phi}_{\text{FR}} = \emptyset \\
 & \qquad \wedge\, \exists\, L \in \widehat{\psi}_{\text{SH}}\ \ \left(L = \bigcup_{i=1}^{k} N_i\right) \wedge\ N_1, N_2 \in \widehat{\text{SH}}\ \ )
\end{array}
$$

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 literals 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 such 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.

We proceed by analyzing what to do when the conditions to be checked at compile-time proposed in previous sections are violated. Due to the complexity of the special case when the second literal is pure, here we will only consider the general case. Thus, we concentrate on analyzing how to proceed when each of the conditions of the general case is violated.

4.1. Condition C1 violated

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

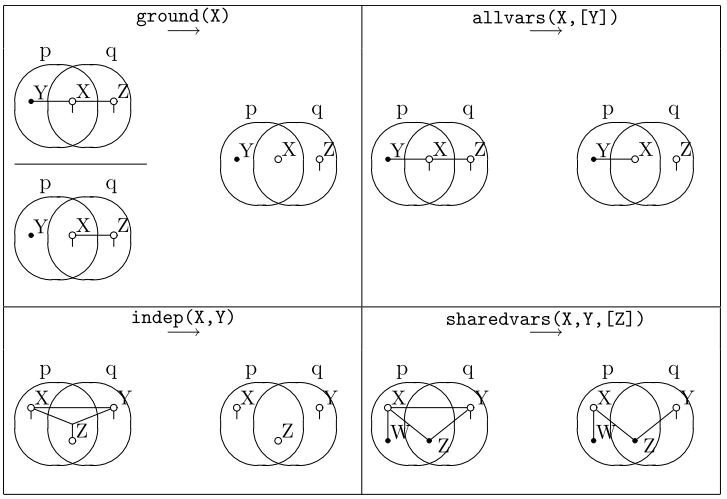


Fig. 5. Example applications of the four checks.

In this case we need run-time checks to ensure that the sharing sets $L \in \widehat{\mathrm{SH}}$ 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 overlapping, it is impossible for all the illegal sharing sets to be inactive, so the goals are definitely not non-strictly independent. 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. In the description of the checks below we say informally "X has property P" to mean that the term to which the variable X has been bound (i.e., $X\theta$) satisfies property P. The order in which such checks 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 $X \in var(p) \cap var(q)$ then we are always 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 the set, such that $X \in var(p)$ and $Y \in var(q)$. 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 $\mathcal{F} = [\,]$.

Fig. 5 gives examples showing how the checks restrict the possible sharing sets.

4.2. Condition C2 violated

$$\begin{aligned}[\;\; & \exists\, \{N_1, N_2, \ldots, N_k\} \subseteq \mathrm{S(p)}, \;\; \text{with } card(X) > 1, \;\; \text{such that} \\ & \forall\, i, j \;\; 1 \le i < j \le k \;\; \Rightarrow \;\; N_i \cap N_j \cap \widehat{\beta}_{\mathrm{FR}} = \emptyset \\ & \wedge \exists\, L \in \widehat{\psi}_{\mathrm{SH}} \;\; \left(L = \bigcup_{i=1}^{k} N_i \right) \wedge N_1, N_2 \in \widehat{\mathrm{SH}} \;\;]. \end{aligned}$$

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.

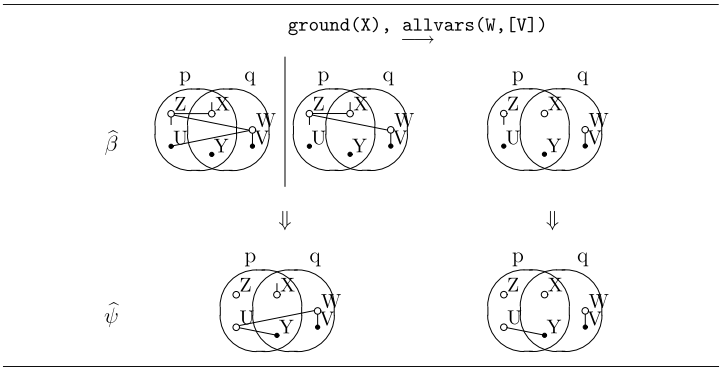


Fig. 6. Restriction of the possible sharing sets by the checks make the goals non-strictly independent.

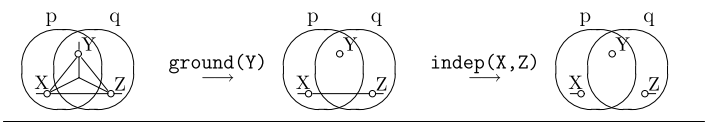


Fig. 7. Restriction of the possible sharing sets performed by the checks.

Now, for each L in the above formula, we compute the different groups of $N_1 \ldots N_k$ that p can join 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 non-strictly independent. 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 expression "`p(X,Y,Z,U) , q(X,Y,W,V)`" and the abstract call and answer substitutions for `p(X,Y,Z,U)` are $\widehat{\beta} = \big(\big[[X]\,[XZ]\,[Y]\,[Z]\,[ZW]\,[U]\,[UW]\,[WV]\big],\,[YUV]\big)$ and $\widehat{\psi} = \big[[X]\,[YU]\,[UW]\,[WV]\big],\,[YV]\big)$. We have that $\widehat{\mathrm{SH}} = \big[[X]\,[XZ]\,[Y]\,[ZW]\,[UW]\big]$, 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 literals, 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 (&-Prolog) parallel operator. Fig. 6 shows the restriction of the possible sharing sets made by the checks, and how this restriction makes the goals non-strictly independent.

4.3. Improved run-time checks for 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 correspond exactly to the conditions traditionally generated for strict independence (shared program variables ground, other program variables independent, see e.g. [33–35] 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 an expression "`p(X,Y) & q(Y,Z)`" with $\widehat{\beta} = \big(\big[[X]\,[Y]\,[Z]\;\;[XY]\,[XZ]\,[YZ]\,[XYZ]\big],\,[\,]\big)$ (i.e., $\top$, equivalent to no information), then we have $\widehat{\mathrm{SH}} = \big[[Y]\,[XY]\,[XZ]\,[YZ]\,[XYZ]\big]$. The check `ground(Y)` covers all the illegal sharing sets except [XZ], which is covered in turn by the check `indep(X,Z)`. Fig. 7 depicts how the checks restrict the possible sharing sets.

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 expression "`p(X,V,W) & q(Y,Z)`" with $\widehat{\beta} = \big(\big[[V]\,[VX]\,[Y]\,[XY]\,[Z]\,[XZW]\,[W]\big],\,[V]\big)$ (see Fig. 8). 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, it is

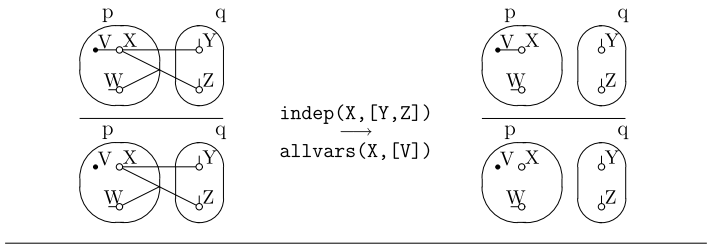


Fig. 8. Restriction of the possible sharing sets performed by either check.

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 `indep(X,Z)` or `indep(W,Z)` tests 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])`.

On the other hand, applying the method presented here, we have that $\widehat{\mathrm{SH}} = \big[[\mathrm{XY}]\,[\mathrm{XZW}]\big]$. 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. Parallelization under NSI: The URLP and CRLP algorithms

We now turn our attention to how, using the conditions stated in previous sections, an automatic parallelizer can insert into the program clauses parallel operators which allow exploiting and-parallelism (the *annotation*).

As mentioned in the introduction, in the traditional approach to annotation, input code is processed by and *Annotator*, or "parallelizer", which explores the bodies of clauses in the given program or module looking for statements and procedure calls which are candidates for parallelization. A number of such algorithms has been proposed [4,5,8,21,30,48,50], including the now classic CDG, UDG, and MEL algorithms [48,50].

However, these annotation algorithms, which were designed with strict independence in mind, are not well suited for parallelization using non-strict independence for a number of reasons. The MEL algorithm can in principle be used for non-strict independence, but it is a simplistic algorithm which cannot create nested parallel expressions, and can thus in some cases miss opportunities for parallelism [48,50]. The CDG and UDG algorithms are more interesting, graph-based algorithms which allow nested parallel expressions. These algorithms may however reorder the literals found to be independent, which is valid for strict independence but not for non-strict independence, because the latter is not symmetric.

As an alternative, we have developed a new algorithm for annotation specially suited to the characteristics of non-strict independence and with simplicity and efficiency in mind. A decision in the design has been to give priority to unconditional parallelism: if we know that two literals are independent without the need for run-time tests, we should always execute them in parallel, even if this prevents the exploitation of further conditional (that is, uncertain) parallelism. The MEL algorithm for strict independence, for example, fails to do this.

The non a priori nature of non-strict independence (and thus the need for correct information from global analysis) combined with the fact that we do not want herein to rerun the analysis on the parallelized expressions (i.e., we would like for simplicity to take as input the analysis results for the sequential program), leads us to another restriction: that *conditional* parallelism will involve only pairs of goals. This is because the run-time tests computed are based on properties about run-time instantiations of program variables in certain points of the program, and thus can only be safely placed in those points. For example, if we wanted to run in parallel three literals, the independence test for the last two is only applicable just before the second one, but we had to include it before the first one. Of course a reanalysis of the program should give the necessary information to compute the test in this case. This is certainly a possibility [31,32], but beyond the scope of this paper.

In fact, this last restriction has some practical repercussions: it keeps the conditional parallel expressions simple, and ensures that, if there exists parallelism between two contiguous atoms, something will run in parallel (this is not true with the CDG or MEL algorithms). The intuition is that it is not worth making complex conditional expressions, which are costly to generate and costly to execute, when we have little information about the runtime values of the variables. Rather, it is preferable to possibly exploit less parallelism in a clause by executing fewer tests than to try to execute the maximum number of literals in parallel in the best case, at the expense of having to run complex tests at run-time and risking losing all the parallelism in the cases that are not so favorable. This intuition is certainly supported by experimental data in the case of strict independence [4,5].

The following two sections present the two steps that comprise the annotator. The first step identifies unconditional parallelism, that is, parallelism that does not need run-time tests. The second step, executed only when run-time tests are allowed (normally controlled by a compiler switch), adds conditional parallelism to the parallel expressions computed by the first step. This guarantees the objective of giving priority to unconditional parallelism over conditional parallelism.

Rule	Pattern	Condition	New Pattern
1	...A, B...	indep(A,B)	...A & B...
2	...PA, B...	(1)	...IA & (DA, B)...
3	...PA, PB...	(2)	...(PA, DB) & IB...

A and B represent single literals and PA, PB, DA, DB, IA and IB represent single literals or parallel expressions.

(1) IA (DA) is the parallel expression formed with the elements of PA from which B is independent (dependent). After the rule is applied, "DA, B" is parallelized recursively. The rule is only applied if IA is not empty.

(2) IB (DB) is the parallel expression formed with the elements of PB which are independent (dependent) from PA. The rule is only applied if IB is not empty.

Fig. 9. Rewriting rules of the URLP algorithm.

5.1. Unconditional annotation: The URLP algorithm

The unconditional annotator, named URLP (Unconditional Recursive Linear Parallelizer), shows a similar degree of parallelization as UDG, but preserving always the original order of the literals in the clause. It is a heuristic algorithm (as CDG), but does not use dependency graphs. Instead it performs a series of rewritings, starting with the sequence of literals of the body of the clause, and applying from left to right the rewriting rules of Fig. 9, in the order in which they are enumerated, until no further changes can be made.

The first rule is obvious: two contiguous literals which are independent can be executed in parallel. The second rule states that when we have a parallel expression followed by a single literal, then that literal can be executed in parallel with the members of the parallel expression from which it is independent, but it must wait until the end of the execution of all goals from which it depends. The third rule is the reciprocal to the second: when we have a parallel expression (or a single literal) followed by a parallel expression, then the members of the last parallel expression which are independent from the first expression can be executed in parallel with it, and the rest must wait until the end of its execution.

Since at each step a simple and correct rule is applied, it can be easily shown that the overall parallelization is correct. To ensure optimality, however, we would need in the general case to provide the execution times of the goals, which is beyond our scope (see [19,20,30,46,47] and their references).

This algorithm has some advantages over UDG: in UDG, in order to compute the dependency graph, every possible dependency between pairs of literals has to be computed. In URLP, however, some checks can be saved, since for example if we have the body clause "a, b, c" and we have found that b depends on a and that c depends on b, we do not need to check the possible dependency between a and c.

As an example of the operation of URLP, consider the parallelization of the body of a clause, which we will schematically represent as "a, b, c, d, e, f, g", where the dependencies given by the conditions of Section 3 are $dep(a, b)$, $dep(b, e)$, $dep(b, f)$, $dep(c, e)$, $dep(d, g)$, and $dep(f, g)$ (we denote by $dep(X, Y)$ that Y is depends on X). The steps followed by the algorithm are shown below:

Step	**Expression**	**Rule used**	**Applied to**
0	a, b, c, d, e, f, g	1	b, c
1	$a, b\,\&c, d, e, f, g$	2	$b\,\&c, d$
2	$a, b\,\&c\,\&d, e, f, g$	2	$b\,\&c\,\&d, e$
3	$a, d\,\&(b\,\&c, e), f, g$	2	$d\,\&(b\,\&c, e), f$
4	$a, d\,\&(b\,\&c, e, f), g$	1	e, f
5	$a, d\,\&(b\,\&c, e\,\&f), g$	3	$a, d\,\&(b\,\&c, e\,\&f)$
6	$(a, b\,\&c, e\,\&f)\,\&d, g$		

In comparison with the final parallel expression above, the parallelization produced by UDG is "$c\,\&d\,\&(a, b, f), e\,\&g$", where the relative order of execution of the literals e and f has been reversed, which, as mentioned before, should be avoided for non-strict independence. Fig. 10 presents the results graphically.

5.2. Conditional annotation: The CRLP algorithm

Once the URLP algorithm has computed an (unconditional) parallel expression, one more step is needed to be able to exploit further parallelism through the use of run-time tests. The two steps comprise what we call the CRLP algorithm (Conditional Recursive Linear Parallelizer).

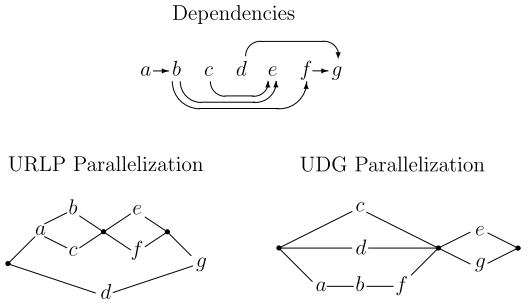


Fig. 10. Example parallelizations using URLP and UDG.

The algorithm examines the input parallel expression to find sequences of literals not yet parallelized, and transforms them inserting the appropriate tests. Let *Exp* be one of these sequences, then **par**(*Exp*) is the transformed expression, defined as:

$$\textbf{par}(\texttt{(p,q,...)}) = \begin{cases} \texttt{p}, \textbf{par}(\texttt{(q,...)}) & \text{if } \textbf{nsi}(\texttt{p},\texttt{q}) = \texttt{false} \\ \texttt{p \& q}, \textbf{par}(\ldots) & \text{if } \textbf{nsi}(\texttt{p},\texttt{q}) = \texttt{true} \\ (\ \textbf{nsi}(\texttt{p},\texttt{q}) \rightarrow \ \texttt{p \& q}, \textbf{par}(\ldots) & \\ \qquad\qquad ;\ \ \texttt{p}, \textbf{par}(\texttt{(q,...)})\ ) & \text{otherwise} \end{cases}$$

where **nsi**(p,q) is the test that ensures non-strict independence between literals p and q (from Section 4.3), but taking into account also the tests already verified by preceding expressions of the branch —this is why **nsi**(p,q) can be just `true`.

It can be easily shown that this simple algorithm meets the restriction that conditional parallelism involves only pairs of literals, and also that it ensures that if there exists parallelism in two contiguous atoms, something will run in parallel.

6. Separating environments: Renaming and replacing variables

When using non-strict independence, and in order to prevent partial answers of a branch that ultimately fails from pruning the search space of other goals to the right, in the theoretical model parallel goals are in principle assumed to run in independent environments (see [34]). When running parallel goals in a distributed environment this separation often occurs naturally since in many implementations the process of sending a goal involves making a copy and performing "back-unification" upon return. However, in many efficient shared-memory implementations goals can often "see" the bindings made by other goals executing concurrently which share variables with them. A simple method for preventing this is to add to the code calls to Prolog's `copy_term` which copy every goal (except one) before running it in parallel and, after the parallel conjunction, unify the original goals and the copied versions, in a similar way to what is done in distributed implementations. However, this solution is inefficient, both in time and memory.

Our objective herein is to optimize the environment separation process, by using the information available from global analysis and the annotation process to pre-compile at least part of the separation operations, eliminating as much as possible unneeded rewritings. Note that simply renaming compile-time variables is not sufficient in general since such variables can be bound at run-time to complex terms containing shared variables. For these cases we will define the following predicate:

`replace_vars(`$[\texttt{X}_1,\ldots,\texttt{X}_n]$`,`$[\texttt{X}'_1,\ldots,\texttt{X}'_n]$`,Z,Z')` :-
$\texttt{Z}'$ *is a term equal to* Z *but with variables* $\texttt{X}'_1,\ldots,\texttt{X}'_n$
in place of variables $\texttt{X}_1,\ldots,\texttt{X}_n$*, respectively*

It can be easily shown that this predicate is generally more efficient than `copy_term`: the latter copies all the term structure (except perhaps, in optimized implementations, some completely ground substructures), whereas the former can also avoid copying non-ground substructures which do not contain variables to be renamed. Furthermore, using `copy_term`, when re-unifying the copies one has to unify the entire goals, whereas using `replace_vars` only the renamed free variables must be re-unified.

We are interested in the potential run-time shared variables, but note that by applying the independence conditions and/or the run-time tests, we ensure that these are the free variables (those of $\widetilde{\beta}_{\text{FR}}$) that appear in the sharing sets of SH (extending this concept to an arbitrary number of literals). Given a goal to be executed in parallel, we can compute the required variable renamings and replacements required for that goal by considering each sharing set of SH as follows:

- If the sharing set does not contain any variables of the goal, nothing must be done.
- If the sharing set contains only one variable of the goal, and the variable is free, only a simple renaming of this variable is needed.
- Else, we need to replace a free variable of the sharing set with a new one in each variable of the goal present in the sharing set (if the goal has free variables in the sharing set, one of them is used to make the replacement).

More precisely, the concrete transformation proceeds as follows:

	$V = \{\mathtt{U},\mathtt{Y}\},\ \mathrm{R}(V) = \{\mathtt{V}\}$			$V = \{\mathtt{W}\},\ \mathrm{R}(V) = \{\mathtt{V},\mathtt{X}\}$		
	$\mathcal{V}$	$\mathcal{R}$	*transformation*	$\mathcal{V}$	$\mathcal{R}$	*transformation*
p(T, V, W)	$\emptyset$	{V}	rv(U, V)	{W}	{V}	ren(W), rv(W, V)
q(U, V, W, X, Y)	{U, Y}	{V}	ren(U), rv(U, Y), rv(U, V)	{W}	{V, X}	ren(W), rv(W, V), rv(W, X)
r(W, Z)	$\emptyset$	$\emptyset$	$\emptyset$	{W}	$\emptyset$	ren(W)

Fig. 11. Environment separation example.

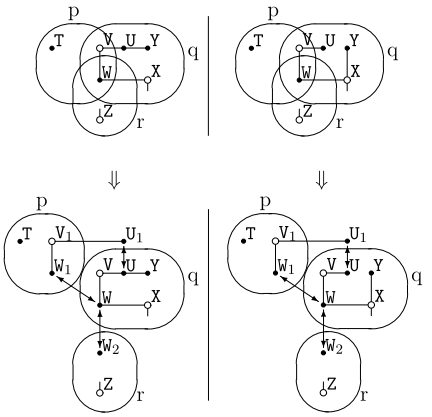


Fig. 12. Representation of the effect of variable replacement in a parallel expression.

- Group into 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 a single element. The motivation is that if two free variables appear in the same sharing set, they are possibly aliased at run-time, and thus they need to be processed together.
- For each of those sets of free shared variables V:
  . Compute $\mathrm{R}(V) = \{w \mid \exists L \in \mathrm{SH}\ \exists v \in V \ \ v \in L \wedge w \in L \wedge w \notin V\}$, i.e., the set containing the variables that appear in the sharing sets of SH with variables from V, excluding those of V. Thus, they possibly contain at run-time variables from V.
  . Then, for each literal p, the necessary renamings or replacements related to V are computed. Let $\mathcal{V} = var(\mathrm{p}) \cap V$ and $\mathcal{R} = var(\mathrm{p}) \cap \mathrm{R}(V)$. We will represent a renaming of a variable v as "ren(v)" and a replacement of a variable v inside w as "rv(v, w)". There are three cases:

    $\mathcal{V} = \emptyset$, $\mathcal{R} = \emptyset \rightarrow$ none.
    $\mathcal{V} = \emptyset$, $\mathcal{R} \neq \emptyset \rightarrow$ rv(v, w) for each $w \in \mathcal{R}$, where $v \in V$.
    $\mathcal{V} \neq \emptyset \rightarrow$ ren(v), rv(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 literals except one, the literal with the most expensive transformation is not considered. Replacements are more expensive than renamings, and replacements in ordinary variables are more expensive than replacements in free variables (which are in fact conditional unifications).
- Once the transformations for all the sets of variables have been computed, then for each literal the replacements in the same variable are joined in a `replace_vars` predicate. Unification ("back-binding") literals must be included after the parallel conjunction for all the free variables renamed or replaced. Note that one side of these unifications is always a free variable, since the conditions ensure that the first literal does not instantiate shared variables.

As an example, consider the parallel expression `p(T,V,W) & q(U,V,W,X,Y) & r(W,Z)`, with the abstract call substitution $\widehat{\beta} = \big(\big[[\mathtt{T}]\,[\mathtt{UV}]\,[\mathtt{UVY}]\,[\mathtt{VWX}]\,[\mathtt{X}]\,[\mathtt{XY}]\,[\mathtt{Z}]\big],\ [TUWY]\big)$. The shared sharing sets are $\mathrm{SH} = \big[[\mathtt{UV}]\,[\mathtt{UVY}]\,[\mathtt{VWX}]\big]$. We have two sets of free variables from SH: {U, Y} and {W}, with $\mathrm{R}(\{\mathtt{U},\mathtt{Y}\}) = \{\mathtt{V}\}$ and $\mathrm{R}(\{\mathtt{W}\}) = \{\mathtt{V},\mathtt{X}\}$. Fig. 11 shows, for each of these sets, and for each literal, the values of $\mathcal{V}$ and $\mathcal{R}$ and the transformation needed. In both columns we discard the transformation for the q/5 literal. The two replacements for the p/3 literal are on the same variable, so they must be joined. Therefore, the parallel expression is transformed into:

```
replace_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
```

Fig. 12 illustrates in our pictorial representation the transformation done. The bidirectional arrows show the bindings performed by the back-binding literals. There are two situations depending on the covering of the free variables by the sharing sets.

```
:- module(qsortdl,[qsort/2],[assertions]).

:- pred qsort/2 : ground * var.

qsort(I,O) :-                   % [[O]],[O]
        qsort(I,O,[]).          % [],[]
qsort([],L,L).
qsort([X|Xs],L,L2) :-           % [[L],[L2],[Sm],[La],[L1]],[L,Sm,La,L1]
        part(Xs,X,Sm,La),       % [[L],[L2],[L1]],[L,L1]
        qsort(Sm,L,[X|L1]),     % [[L,L1],[L2]],[L1]
        qsort(La,L1,L2).        % [[L,L2,L1]],[]
part([],_,[],[]).
part([E|R],C,[E|Sm1],La) :-     % [[La],[Sm1]],[La,Sm1]
        E=<C,                   % [[La],[Sm1]],[La,Sm1]
        !,
        part(R,C,Sm1,La).       % [],[]
part([E|R],C,Sm,[E|La1]) :-     % [[Sm],[La1]],[Sm,La1]
        E>C,                    % [[Sm],[La1]],[Sm,La1]
        part(R,C,Sm,La1).       % [],[]
```

Fig. 13. Example program to be parallelized.

7. Example parallelization of a program

As an example, in this section we show how to apply the proposed methods to a concrete program (quick-sort using difference lists) in order to exploit the non-strictly independent and-parallelism it contains. Although the program is small, we think that it illustrates the techniques proposed, and it is compact enough to allow presenting the information inferred by the *Sharing+Freeness* analyzer and the resulting parallelization process. The program is listed in Fig. 13. Again, the Ciao `pred` assertion defines the mode of use. This can often be inferred through modular analysis from the module calling `qsort/2` in this `qsortdl` module, making the assertion potentially unnecessary. The abstract states actually inferred at each program point by the Ciao preprocessor using the *Sharing+Freeness* domain are shown as comments.

We will concentrate on the parallelization of the `qsort/3` predicate. First, we will analyze whether it is possible to parallelize the first and second literals of the recursive clause of `qsort/3`, so we have that p = `part(Xs,X,Sm,La)` and q = `qsort(Sm,L,[X|L1])`, and the abstract substitutions involved are $\widehat{\beta} = \big(\big[[L]\,[L2]\,[Sm]\,[La]\,[L1]\big],\,[L\,Sm\,La\,L1]\big)$ and $\widehat{\psi} = \big(\big[[L]\,[L2]\,[L1]\big],\,[L\,L1]\big)$. Then, we compute the set $\widehat{SH} = \big[[Sm]\big]$. Condition C1 is not met, since "Sm" is not in $\widehat{\psi}_{FR}$, and furthermore this is a free variable that does not appear in another sharing set in $\widehat{\beta}_{SH}$, 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 literals of the clause are not non-strictly independent either.

Finally, let us try with the second and third literals in the same clause. Now p = `qsort(Sm,L,[X|L1])`, q = `qsort(La,L1,L2)`, $\widehat{\beta} = \big(\big[[L]\,[L2]\,[L1]\big],\,[L\,L1]\big)$ and $\widehat{\psi} = \big(\big[[L\,L1]\,[L2]\big],\,[L1]\big)$. The shared sharing sets are $\widehat{SH} = \big[[L1]\big]$. But now the conditions hold: L1 $\in \widehat{\psi}_{FR}$ and no sharing sets meet ¬C2. Thus in this case we have non-strict independence, and no run-time checks are needed (note also that the literals are not strictly independent, since they share the free variable "L1"). A simple renaming of L1 ensures environment separation (although it is not strictly needed in this case). The URLP parallelizer thus turns the original `qsort/3` predicate definition into:

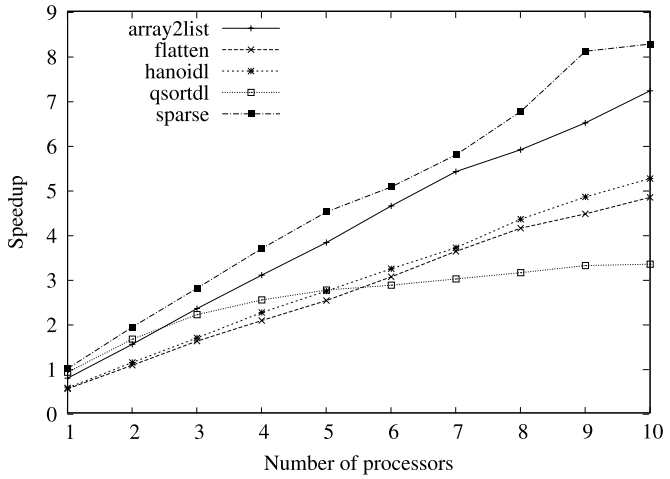
```
qsort([],L,L).
qsort([X|Xs],L,L2) :- part(Xs,X,Sm,La),
                      qsort(Sm,L,[X|L1P]) & qsort(La,L1,L2), L1 = L1P
```

where "&" is again the (unconditional) parallel operator.

8. Implementation and some experimental results

An exhaustive study of the amount of non-strict and-parallelism that can be obtained from real programs using these techniques is beyond the scope of this paper. However, we include for completeness the results from a number of experiments performed in [6] with a small number of programs that have literals which are non-strictly independent but no literals which are strictly independent. The programs used in this experiment are:

- **array2list**: a subroutine of the SICStus Prolog "arrays.pl" library which translates an extensible array into a list of index-element pairs. The input array used to measure the speedups had 2048 elements.
- **flatten**: the subroutine shown in the introduction which flattens a list of lists of any nesting depth into a plain list. The speedups were measured with an input list of 987 elements with a recursive "depth" of 7.
- **hanoi_dl**: 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.



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_dl	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

Fig. 14. Speedups of several programs with non-strict independence.

- **qsort_dl**: the quick-sort sorting algorithm using difference lists of Section 7. The speedups were measured sorting a list of 300 elements.
- **sparse**: 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.

In order to generate these results the relevant parts of the &-Prolog parallelizing compiler were modified to be able to parallelize programs using non-strict independence and *Sharing+Freeness* information. This implied essentially adding the URLP/CRLP algorithms to the set of "annotators" supported (plus adding the new `replace_vars` library predicate, the new run-time tests, etc. to the libraries to be included at run time) and implementing the conditions of Section 3. These modifications were later incorporated into the Ciao preprocessor. In all cases unconditional parallelism was generated (i.e., no run-time checks were necessary for these programs).

Using this setup the programs were automatically parallelized and then executed using from 1 to 10 processors on the previously mentioned &-Prolog system [10,26,29], an efficient parallel implementation of full Prolog capable of exploiting and-parallelism among (possibly non-deterministic) goals. The experiments were run on a Sequent Symmetry which, while a slow machine by today's standards, is quite related architecturally to modern multicore servers such as, for example, an 8-core Sun Fire T2000. The results are given in Fig. 14. Speedups are relative to the execution time on one processor of the original (unparallelized) program. The slight slowdown exhibited by the parallelized program on one processor w.r.t. the sequential program is due to the (small) overhead introduced by the parallel conjunctions (`&`) with respect to sequential sequencing [35]. More concretely, the performance on one processor of the &-Prolog version used in these tests, when running a parallelized program, is about 95% of the performance of the sequential system on which it is based (an early version of SICStus Prolog [7]), itself a high-performance and popular Prolog system.

Since the results improve significantly on the raw speed of the sequential program on a competitive Prolog implementation it can be argued that the speedups are meaningful and useful. Note that none of these programs obtains any speedup if parallelized using strict independence. Again, it is beyond the scope of this paper to study the amount of non-strict and-parallelism that can be obtained from real programs using the proposed techniques, but we include these early results because we believe that they are encouraging.

9. Conclusions and future work

While "non-strict" independence offers clear advantages over "strict" independence in terms of generality and the amount of parallelism it can exploit, its compile-time/run-time detection is also clearly more involved. The techniques that

we have presented are based on the availability of certain information about run-time instantiations of program variables – sharing and freeness – for the inference of which a significant amount of compile-time technology is available. We have presented techniques for combined compile-time/run-time detection of non-strict independence, proposing specialized classes of run-time checks for this type of parallelism as well as algorithms for implementing such checks. We have also proposed the URLP/CRLP annotation (parallelization) algorithms, which are specially suited to non-strict independence as well as novel ways of using the *Sharing+Freeness* information to optimize how the run-time environments of goals are kept apart during parallel execution. An example of the application of the proposed methods to a concrete program has also been given, along with the output as obtained by our integrated compiler (now within CiaoPP) which combines the techniques proposed with a *Sharing+Freeness* analyzer, and the URLP/CRLP-based parallelizer (annotator). We have also included some early experimental speedup results obtained for this and other programs presenting non-strict independence.

Our work makes it possible also to understand more clearly in what way the *Sharing+Freeness* analysis itself can be improved to increment the amount of parallelism that can be exploited automatically. The 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 [60] (among others [2,12,14,37,38,44,61]). Some such combinations have been incorporated in the past in our system by using the techniques described in [11], 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 still limited by the fact that the sharing and freeness information remains restricted to variables which appear in the program. Additional improvements can 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 instantiation states of the variables which appear inside these terms. To achieve this goal, sharing and freeness can be integrated (by using again the techniques of [11] or [15]) with other analyses, like the depth-k [43,57] domain, or, even better, "pattern" [15] or ultimately recursive type analyses (e.g., [3,22,62]). We expect the approach presented for simple *Sharing+Freeness* to be still valid directly or with small modifications for these more sophisticated types of analyses. For example, if a depth-k analysis is used in combination with *Sharing+Freeness*, the information on sharing and freeness obtained will then refer to the variables in the depth-k terms, but the same parallelization conditions presented in this paper would apply, by applying them to those new variables and their sharing and freeness information before and after execution of goals.

Acknowledgments

The authors would like to thank M. Bruynooghe for suggesting improvements to the first version of the C1 condition of Section 3.1 and E. Zaffanella and another (anonymous) reviewer of this journal version for numerous suggestions that have improved both the content and the presentation. The research presented in this paper (both the early contributions and the later extensions for this journal version) have been supported in part by the Information Society Technologies program of the European Commission under projects "PARFORCE", "ACCLAIM", and "MOBIUS", by the Spanish Ministry of Education under the *IPL-D*, *MERIT*, and *DOVES* projects, and by the Madrid Regional Government under the *PROMESAS* program. Manuel Hermenegildo was also supported in part by the Prince of Asturias Chair in Information Science and Technology at the U. of New Mexico. As mentioned before, this paper is an extended version of an earlier paper by the same authors which was published in the 1994 Static Analysis Symposium [6].

References

[1] G. Almasi, A. Gottlieb (Eds.), Highly Parallel Computing, Benjamin Cummins, San Francisco, CA, 1994.
[2] M. Bruynooghe, M. Codish, A. Mulkers, Abstract unification for a composite domain deriving sharing and freeness properties of program variables, in: F.S. de Boer, M. Gabbrielli (Eds.), Verification and Analysis of Logic Languages, 1994, pp. 213–230.
[3] M. Bruynooghe, G. Janssens, An instance of abstract interpretation integrating type and mode inference, in: Fifth International Conference and Symposium on Logic Programming, Seattle, Washington, MIT Press, August 1988, pp. 669–683.
[4] F. Bueno, M. García de la Banda, M. Hermenegildo, Effectiveness of global analysis in strict independence-based automatic program parallelization, in: 1994 International Symposium on Logic Programming, 1993.
[5] F. Bueno, M. García de la Banda, M. Hermenegildo, Effectiveness of abstract interpretation in automatic parallelization: A case study in logic programming, ACM TOPLAS 21 (2) (1999) 189–238.
[6] D. Cabeza, M. Hermenegildo, Extracting non-strict independent and-parallelism using sharing and freeness information, in: 1994 International Static Analysis Symposium, Namur, Belgium, in: LNCS, vol. 864, Springer-Verlag, September 1994, pp. 297–313.
[7] M. Carlsson, Sicstus Prolog User's Manual. Po Box 1263, S-16313 Spanga, Sweden, February 1988.
[8] A. Casas, M. Carro, M. Hermenegildo, Annotation algorithms for unrestricted independent and-parallelism in logic programs, in: 17th International Symposium on Logic-based Program Synthesis and Transformation, LOPSTR'07, The Technical University of Denmark, in: LNCS, vol. 4915, Springer-Verlag, August 2007, pp. 138–153.
[9] A. Casas, M. Carro, M. Hermenegildo, A high-level implementation of non-deterministic, unrestricted, independent and-parallelism, in: M. García de la Banda, E. Pontelli (Eds.), 24th International Conference on Logic Programming, ICLP'08, in: LNCS, Springer-Verlag, December 2008.
[10] A. Casas, M. Carro, M. Hermenegildo, Towards a high-level implementation of execution primitives for non-restricted, independent and-parallelism, in: D.S. Warren, P. Hudak (Eds.), 10th International Symposium on Practical Aspects of Declarative Languages, PADL'08, in: LNCS, vol. 4902, Springer-Verlag, 2008, pp. 230–247.
[11] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, M. Hermenegildo, Improving abstract interpretations by combining domains, ACM Transactions on Programming Languages and Systems 17 (1) (1995) 28–44.
[12] Michael Codish, Dennis Dams, Gilberto Filé, Maurice Bruynooghe, On the design of a correct freeness analysis for logic programs, The Journal of Logic Programming 28 (3) (1996) 181–206.

[13] J.S. Conery, The And/Or process model for parallel interpretation of logic programs, Ph.D. Thesis, The University of California At Irvine, 1983, Technical Report 204.
[14] A. Cortesi, 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, New York, 1991, pp. 52–61.
[15] A. Cortesi, B. Le Charlier, P. Van Hentenryck, Combinations of abstract domains for logic programming, in: POPL'94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, ACM, January 1994, pp. 227–239.
[16] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Fourth ACM Symposium on Principles of Programming Languages, 1977, pp. 238–252.
[17] M. García de la Banda, Independence, global analysis, and parallelism in dynamically scheduled constraint logic programming, Ph.D. Thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid, Spain, September 1994.
[18] M. García de la Banda, M. Hermenegildo, K. Marriott, Independence in CLP languages, ACM Transactions on Programming Languages and Systems 22 (2) (2000) 269–339.
[19] S. K. Debray, N.-W. Lin, M. Hermenegildo, Task granularity analysis in logic programs, in: Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation, ACM Press, June 1990, pp. 174–188.
[20] S.K. Debray, P. López-García, M. Hermenegildo, N.-W. Lin, Estimating the computational cost of logic programs, in: Static Analysis Symposium, SAS'94, Namur, Belgium, in: LNCS, vol. 864, Springer-Verlag, September 1994, pp. 255–265.
[21] D. DeGroot, Restricted AND-Parallelism, in: International Conference on Fifth Generation Computer Systems, Tokyo, November 1984, pp. 471–478.
[22] J. Gallagher, K. Henriksen, Abstract domains based on regular types, in: B. Demoen, V. Lifschitz (Eds.), ICLP 2004, Proceedings, in: LNCS, vol. 3132, Springer-Verlag, 2004, pp. 27–42.
[23] G. Gupta, B. Jayaraman, Compiled and-or parallelism on shared memory multiprocessors, in: 1989 North American Conference on Logic Programming, MIT Press, October 1989, pp. 332–349.
[24] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, M. Hermenegildo, Parallel execution of prolog programs: A survey, ACM Transactions on Programming Languages and Systems 23 (4) (2001) 472–602.
[25] S. Haridi, S. Janson, Kernel andorra prolog and its computation model, in: D.H.D. Warren, P. Szeredi (Eds.), Proceedings of the Seventh International Conference on Logic Programming, MIT Press, June 1990, pp. 31–46.
[26] M. Hermenegildo, An abstract machine for restricted AND-parallel execution of logic programs, in: Third International Conference on Logic Programming, Imperial College, in: Lecture Notes in Computer Science, vol. 225, Springer-Verlag, July 1986, pp. 25–40.
[27] M. Hermenegildo, Parallelizing irregular and pointer-based computations automatically: Perspectives from logic and constraint programming, Parallel Computing 26 (13–14) (2000) 1685–1708.
[28] M. Hermenegildo, The CLIP Group, Some methodological issues in the design of CIAO - a generic, parallel, concurrent constraint system, in: Principles and Practice of Constraint Programming, in: LNCS, vol. 874, Springer-Verlag, May 1994, pp. 123–133.
[29] M. Hermenegildo, K. Greene, The &-Prolog system: Exploiting independent and-parallelism, New Generation Computing 9 (3–4) (1991) 233–257.
[30] M. Hermenegildo, G. Puebla, F. Bueno, P. López-García, Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor), Science of Computer Programming 58 (1–2) (2005) 115–140.
[31] M. Hermenegildo, G. Puebla, K. Marriott, P. Stuckey, Incremental analysis of logic programs, in: International Conference on Logic Programming, MIT Press, June 1995, pp. 797–811.
[32] M. Hermenegildo, G. Puebla, K. Marriott, P. Stuckey, Incremental analysis of constraint logic programs, ACM Transactions on Programming Languages and Systems 22 (2) (2000) 187–223.
[33] M. Hermenegildo, F. Rossi, On the correctness and efficiency of independent And-Parallelism in logic programs, in: 1989 North American Conference on Logic Programming, MIT Press, October 1989, pp. 369–390.
[34] M. Hermenegildo, F. Rossi, Non-strict independent and-parallelism, in: 1990 International Conference on Logic Programming, MIT Press, June 1990, pp. 237–252.
[35] M. Hermenegildo, F. Rossi, Strict and non-strict independent and-parallelism in logic programs: Correctness, efficiency, and compile-time conditions, Journal of Logic Programming 22 (1) (1995) 1–45.
[36] M. Hermenegildo, R. Warren, Designing a high-performance parallel logic programming system, Computer Architecture News, Special Issue on Parallel Symbolic Programming 15 (1) (1987) 43–53.
[37] P. M. Hill, R. Bagnara, E. Zaffanella, Soundness, idempotence and commutativity of set-sharing, Theory and Practice of Logic Programming 2 (2) (2002) 155–201.
[38] P. M. Hill, E. Zaffanella, R. Bagnara, A correct, precise and efficient integration of set-sharing, freeness and linearity for the analysis of finite and rational tree languages, Theory and Practice of Logic Programming 4 (3) (2004) 289–323.
[39] D. Jacobs, A. Langen, Accurate and efficient approximation of variable aliasing in logic programs, in: 1989 North American Conference on Logic Programming, MIT Press, October 1989.
[40] D. Jacobs, A. Langen, Static analysis of logic programs for independent and-parallelism, Journal of Logic Programming 13 (2–3) (1992) 291–314.
[41] S. Janson, S. Haridi, Programming paradigms of the andorra kernel language, in: 1991 International Logic Programming Symposium, MIT Press, 1991, pp. 167–183.
[42] A.H. Karp, R.C. Babb, A comparison of 12 parallel fortran dialects, IEEE Software (1988).
[43] A. King, P. Soper, Depth-k sharing and freeness, in: International Conference on Logic Programming, MIT Press, June 1994.
[44] Giorgio Levi, Fausto Spoto, Non pair-sharing and freeness analysis through linear refinement, in: Partial Evaluation and Semantic-Based Program Manipulation, 2000, pp. 52–61.
[45] Xuan Li, Andy King, Lunjin Lu, Lazy set-sharing analysis, in: Philip Wadler, Masimi Hagiya (Eds.), 8th. Int'l. Symp. on Functional and Logic Programming, in: LNCS, Springer-Verlag, April 2006.
[46] P. López-García, M. Hermenegildo, S.K. Debray, A methodology for granularity based control of parallelism in logic programs, Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation 21 (4–6) (1996) 715–734.
[47] E. Mera, P. López-García, M. Carro, M. Hermenegildo, Towards execution time estimation in abstract machine-based languages, in: 10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP'08, ACM Press, July 2008, pp. 174–184.
[48] K. Muthukumar, F. Bueno, M. García de la Banda, M. Hermenegildo, Automatic compile-time parallelization of logic programs for restricted, goal-level, independent and-parallelism, Journal of Logic Programming 38 (2) (1999) 165–218.
[49] K. Muthukumar, M. Hermenegildo, Determination of variable dependence information at compile-time through abstract interpretation, in: 1989 North American Conference on Logic Programming, MIT Press, October 1989, pp. 166–189.
[50] K. Muthukumar, M. Hermenegildo, The CDG, UDG, and MEL methods for automatic compile-time parallelization of logic programs for independent and-parallelism, in: Int'l. Conference on Logic Programming, MIT Press, June 1990, pp. 221–237.
[51] K. Muthukumar, M. Hermenegildo, Combined determination of sharing and freeness of program variables through abstract interpretation, in: 1991 International Conference on Logic Programming, MIT Press, June 1991, pp. 49–63.
[52] K. Muthukumar, M. Hermenegildo, Compile-time derivation of variable dependency using abstract interpretation, Journal of Logic Programming 13 (2/3) (1992) 315–347.
[53] J. Navas, F. Bueno, M. Hermenegildo, Efficient top-down set-sharing analysis using cliques, in: Eight International Symposium on Practical Aspects of Declarative Languages, in: LNCS, vol. 2819, Springer-Verlag, January 2006, pp. 183–198.
[54] E. Pontelli, G. Gupta, M. Hermenegildo, &ACE: A high-performance parallel Prolog system, in: International Parallel Processing Symposium, in: IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995, pp. 564–572.

[55] E. Pontelli, G. Gupta, D. Tang, M. Carro, M. Hermenegildo, Improving the efficiency of nondeterministic and–parallel systems, The Computer Languages Journal 22 (2/3) (1996) 115–142.
[56] B. Ramkumar, L.V. Kale, Compiled execution of the reduce-OR process model on multiprocessors, in: S. Debray, M. Hermenegildo (Eds.), 1989 North American Conference on Logic Programming, MIT Press, October 1989, pp. 313–331.
[57] T. Sato, H. Tamaki, Enumeration of success patterns in logic programs, Theoretical Computer Science 34 (1984) 227–240.
[58] K. Shen, Exploiting dependent and-parallelism in Prolog: The dynamic, dependent and-parallel scheme, in: Proc. Joint Int'l. Conf. and Symp. on Logic Prog., MIT Press, 1992, pp. 717–731.
[59] K. Shen, Studies in And/Or parallelism in Prolog, Ph.D. Thesis, U. of Cambridge, 1992.
[60] H. Søndergaard, An application of abstract interpretation of logic programs: Occur check reduction, in: European Symposium on Programming, in: LNCS, vol. 123, Springer-Verlag, 1986, pp. 327–338.
[61] R. Sundarajan, J.S. Conery, An abstract interpretation scheme for groundness, freeness, and sharing analysis of logic programs, in: 12th FST & TCS Conference, New Delhi, India, December 1992.
[62] C. Vaucheret, F. Bueno, More precise yet efficient type inference for logic programs, in: International Static Analysis Symposium, in: LNCS, vol. 2477, Springer-Verlag, September 2002, pp. 102–116.
[63] D.H.D. Warren, The extended andorra model with implicit control, in: Sverker Jansson (Ed.), Parallel Logic Programming Workshop, Box 1263, S-163 13 Spanga, SWEDEN, SICS, June 1990.
[64] E. Zaffanella, R. Bagnara, P.M. Hill, Widening sharing, in: G. Nadathur (Ed.), Principles and Practice of Declarative Programming, in: Lecture Notes in Computer Science, vol. 1702, Springer-Verlag, Berlin, 1999, pp. 414–432.