Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming

M. García de la Banda

FIM/87.1/DIA/95

Facultad de Informática
Universidad Politécnica de Madrid
Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming

María García de la Banda

The notion of independence is crucial in and-parallel execution, since it is a pre-condition for ensuring the correctness and usefulness of such parallelism with respect to the associated sequential execution. It is also at the heart of other optimizations such as goal reordering and intelligent backtracking. This dissertation builds on previous work in the context of independence-based and-parallelism for logic programs. Its objective is to extend the applicability of such work to the more general class of constraint logic programs, and also of constraint logic programs with dynamic scheduling. In doing this we first extend the concept of independence to constraint logic programming, give sufficient conditions that are easier to detect at run-time than the definition of independence, propose a simple parallel model, prove the correctness and efficiency of the approach, and show the strong relationship between correctness, efficiency, and search space preservation. We also show how generalizing independence to arbitrary constraint logic languages and constraint solvers yields new insights into independence even for standard logic programs. We also study the importance of the concept of independence for other applications, including reordering and intelligent backtracking. We then perform a further generalization of the notion of independence in constraint logic programs to the context of constraint logic programs with dynamic scheduling.

Having defined the notions of independence needed, and proposed sufficient run-time tests, we address the issue of its static detection. We start by performing an exhaustive study aimed at assessing the usefulness of data-flow analysis tools in the task of compile-time automatic parallelization of logic programs, based on strict independence detection. We evaluate the trade-offs in precision and analysis time and also the effectiveness of the combined domains in program parallelization. Our overall conclusion, based on the improvements observed, is that global analysis based on abstract interpretation is indeed a powerful tool in this application. Motivated by the results obtained from our studies, we generalize previously proposed global data-flow analysis techniques for logic programs to the context of constraint logic languages, and to constraint logic languages with dynamic scheduling. In particular, we propose and report on the implementation of two complete frameworks for the analysis of constraint logic languages and dynamically scheduled languages, respectively. These extensions provide the basis not only for reducing the run-time overhead of independence detection, but also other optimizations which remove the overhead of constraint solving and dynamic scheduling and have the potential to provide high performance to constraint logic languages with dynamic scheduling even in a sequential implementation.
To my parents.
Acknowledgments

Since I can remember, I wanted to be a researcher. My father was a researcher and I inherited his passion for something which I thought was the most important work in the world (well, perhaps with being an astronaut). However, time passed and I found myself working as a Cobol programmer for Siemens, a thousand miles way from my aims. One day, Manuel Hermenegildo offered me to start working at the University and join the CLIP lab. I did not doubt a second. However, I have many other things to thank him for. He gave me all the support and resources one could wish. He gave me his knowledge, his intuition for determining where the real problem is, and how to solve it, and his time for discussing any single issue I could have doubts about. He corrected my work patiently, taught me how to improve it, encouraged me to report the results of my research, perhaps the part that I like less, and supported me when the rejections came. This thesis would never have been even started without him. I have also to thank Francisco Bueno, the first person I worked with; my colleague and friend during all these happy years. He opened my mind to different points of view, helped me in implementing many of the systems presented in this thesis, and was always ready to clarify my doubts with his knowledge and analysis ability. We have shared so many nights working at the computer that now I cannot imagine working without him. I would like also to thank my parents; they alway trusted me and supported me in all decisions I have ever taken, even when they disagreed. Their love, support, and comprehension have made it possible for many good things to happen, including finishing this thesis. Thanks also to all my other colleagues at the CLIP lab: Manuel Carro, Daniel Cabeza, Pedro López, Germán Puebla, and María José Fernández, not only for their continuous help, but also for their friendship and happiness which helped me when I was depressed. I am also indebted to Kim Marriott, Maurice Bruynooghe, and Michael Codish with whom I had the pleasure to collaborate. They gave me their time, their advice, their knowledge, and taught me that a good presentation, explanation, and formalization of the work is as important as a good idea. I also thank Peter Stuckey and Harald Søndergaard with whom I had many clarifying discussions and who gave me precious comments about my work. Francesca Rossi and Ugo Montanari who introduced me to new fields. Kish
Shen who provide me with many useful comments and suggestions for improving my work. Giorgio Levi, Roberto Giacobazzi and Moreno Falaschi, with whom I had very stimulating discussions. John Gallagher, Andre de Waal, Gerda Janssens, Veroniek Dumortier, Wim Simoens, and all the people with whom I have collaborated in the ESPRIT and CICYT projects, in the context of which this work has been developed. Finally, I am indebted to ESPRIT, CICYT, and the Spanish Ministry of Science and Education for the funds that have made this work possible.
Contents

1 Introduction ........................................ 1
   1.1 Parallelism and Logic Programming ................. 3
   1.2 And-Parallelism .................................. 5
   1.3 Independence, Automatic Parallelization and Data-Flow Analysis .... 9
   1.4 Research Goals .................................. 11
   1.5 Thesis Overview ................................ 12
   1.6 Main Contributions ................................ 14

2 Background ........................................... 17
   2.1 Basic Definitions ................................ 17
   2.2 Logic Programming ................................. 18
   2.3 Abstract Interpretation ............................. 23

II Independence and Parallelism .......................... 27

3 Independence in Constraint Logic Programming Languages .... 29
   3.1 Background ....................................... 31
   3.2 Independence and Search Space Preservation ........... 36
      3.2.1 Independence in Logic Programs ............... 36
      3.2.2 Independence in Constraint Logic Programs .... 42
   3.3 Levels of Independence ............................. 48
      3.3.1 Weak Independence ............................. 48
      3.3.2 Strong Independence ........................... 50
      3.3.3 Search Independence ........................... 52
   3.4 Ensuring Search Independence “A Priori” .............. 53
   3.5 Solver Independence ................................. 57
   3.6 Applications ...................................... 60
      3.6.1 Independent And-Parallelism .................... 61
5.3.5 Combined Domains ............................................. 113
5.4 IDRA: Ideal Resource Allocation ................................. 113
5.5 Experimental Results ............................................. 115
  5.5.1 Benchmark Programs ........................................ 116
  5.5.2 Efficiency Results .......................................... 116
  5.5.3 Effectiveness Results: Static Tests ......................... 119
  5.5.4 Effectiveness Results: Dynamic Tests ...................... 121
5.6 Discussion ...................................................... 122
5.7 Chapter Conclusions ............................................ 127

6 Analyzing Constraint Logic Programs ............................ 129
  6.1 Towards a CLP Analysis Framework ............................. 130
  6.2 Modifying the CLP Operational Semantics .................... 132
  6.3 Extension of the Analysis Framework ......................... 134
    6.3.1 The Abstract Domain .................................... 134
    6.3.2 The Abstract Operations ................................ 136
    6.3.3 Improving the abstract_exit Function .................... 138
    6.3.4 Passive Constraints ..................................... 141
  6.4 Inference of Definiteness Information ....................... 143
    6.4.1 Abstract Domain and Abstraction Function ............... 145
    6.4.2 Abstract Projection and Abstract Conjunction Functions 148
  6.5 Experimental Results ......................................... 150
    6.5.1 Benchmark Programs ..................................... 151
    6.5.2 Results and Discussion .................................. 152
  6.6 Chapter Conclusions .......................................... 155

7 Analyzing Logic Programs with Dynamic Scheduling ............... 157
  7.1 Operational Semantics ........................................ 159
  7.2 Abstract Interpretation ....................................... 161
  7.3 Denotational Semantics ...................................... 162
  7.4 Implementation .............................................. 167
  7.5 Termination .................................................. 168
  7.6 Performance Results ......................................... 169
  7.7 Chapter Conclusions .......................................... 172

8 Conclusions and Future Work ..................................... 175
  8.1 And-parallelism – A Unifying View ............................ 176
  8.2 Topics for Future Work ....................................... 180
Chapter 1

Introduction

Among existing programming paradigms, in this thesis we concentrate on logic programming \cite{97,34} and its recent offsprings constraint logic programming \cite{86} and its dynamically scheduled variants \cite{147,56,25,127,21,35,68,55}. From the problem point of view, logic programming, with its built-in search facilities, the power of the logical variable, a powerful set of dynamic data structures, and unification as a generalization of pattern matching, offers ease of programming for complex applications with a strong symbolic component. The two main perceived drawbacks of logic programming languages – speed and interfacing – have been minimized by recent progress in implementation techniques. Prolog \cite{34} compilers are available for all standard operating systems and interface well to these operating systems and to modules written in other programming languages. From the performance point of view recent, WAM-based \cite{151}, implementations offer quite good performance, certainly within an order of magnitude of lower-level programming languages such as C, and often, when recently proposed optimizations are applied, within a factor of two of such languages \cite{133,134,146}. Furthermore, it has recently been shown \cite{144,3,72} that parallel logic programming systems can be built which preserve the speed of current sequential implementations of logic languages in each processor, and can achieve an order of magnitude speedup on commercial multiprocessors. This already represents a faster execution than C. Of course, parallelization of C would also achieve greater speed, but such parallelization is simpler for a higher-level language.

The constraint logic programming (CLP) paradigm \cite{86} is a relatively recent proposal which has emerged as the natural combination of the constraint solving and logic programming paradigms. This combination makes it possible to parameterize a logic-based language and its semantics by a choice of domain of computation and constraints. In this context, traditional logic programming (LP) can be seen as an instance of CLP in which constraints are equations over terms and the domain of computation is
the traditional Herbrand domain with the well known unification algorithm [132, 113].
Thus, constraint logic programming significantly extends the expressive power of logic
programming to, for example, finite domain and numerical problems, previously the
turf of imperative languages.

Another field which has caught the interest of the research community is the class
of second-generation logic programming languages, such as IC-Prolog [25], MU-Prolog
[127], NU-Prolog [147], Prolog-II [56], Sicstus-Prolog [21], Prolog-III [35], CHIP [68],
SEPIA [55], etc. These languages provide a more flexible form of goal scheduling in
which computation generally proceeds left-to-right but in which some calls are dynam-ically “delayed” until their arguments are sufficiently instantiated to allow the call to
run efficiently. Such dynamic scheduling overcomes many problems associated with tra-
ditional Prologs and their fixed scheduling. First, it allows the same program to have
many different and efficient operational semantics as the operational behavior depends
on which arguments are supplied in the query. Thus, programs really behave efficiently
as relations, rather than as functions. Second, the treatment of negation is sound, as
negative calls are delayed until all arguments are ground. Third, it allows intelligent
search in combinatorial constraint problems. Finally, dynamic scheduling allows a new
style of programming in which procedures are viewed as processes which communicate
asynchronously through shared variables.

It is important to note that the two extensions mentioned in the previous paragraphs
are largely orthogonal. This becomes clear when considering logic languages as the
combination of two components: the logic component modeling the semantics of “pure”
programs, and the control component specifying the computation procedure under a
specific control-flow strategy. While the CLP paradigm affects the former, dynamic
scheduling affects the latter. Thus, both extensions can coexist, as exemplified by
many real languages [56, 35, 68].

Unfortunately, both extensions have performance problems. The efficiency of CLP
programs can be better than that obtained by an equivalent program written in a
more conventional language when the program can take advantage of constraint-based
pruning of the search space. However, CLP systems are still slower than required for
many real applications, due both to the high number of constraints to solve and to
the cost of the constraint solver algorithms. Additionally, the expressive power is paid
for in terms of efficiency with respect to traditional logic programming systems when
running typical (i.e., Prolog) logic programs. Dynamic scheduling also implies a non-
trivial cost. Goals, if affected by a delay declaration, must be checked to see whether
they should delay or not. Variables upon whose values the execution of a goal depends
must be tagged accordingly. Upon variable binding, it is possible that delayed calls
must be woken or put in a “pending” list, so that they are woken before the next goal is executed. Also, few register allocation optimizations can be performed for delayed goals. Finally, additional space needs to be allocated for delayed goals until they are woken [20].

Such performance limitations, combined with the increasing acceptance of these languages, have motivated a growing interest in the application to such languages of the many optimization techniques which have been shown to be useful in the context of LP. The objective of this thesis is to extend the applicability of such optimizations, and in particular the parallelization techniques, to the context of dynamically scheduled constraint logic programming languages. In doing this, and since it seems that the effectiveness of most of such optimizations depends on the information gathered by data-flow analyses, we will devote a significant part of the thesis to the extension of data-flow analysis techniques, and in particular those based on abstract interpretation, to the new kinds of languages being considered.

1.1 Parallelism and Logic Programming

The two main types of parallelism which can be exploited in logic programs are well known[36]: (1) and-parallelism and (2) or-parallelism. Several models have been proposed to take advantage of such opportunities (see, for example, [54], [131], [8], [78], [99], [156], [64], [153], [144], [3] and their references). The following classical example and Figure 1.1 (taken from [81]) serves to illustrate where parallelism is available:

\[
\begin{align*}
\text{crew}(x,y) & \leftarrow \text{navigator}(x), \text{pilot}(y). \\
\text{crew}(x,y) & \leftarrow \text{mechanic}(x), \text{pilot}(y).
\end{align*}
\]

As can be seen in Figure 1.1 there are two alternative ways to try to satisfy the goal crew(peter,z) corresponding to the two clauses in the definition of crew/2. It is possible to have different processors proceed simultaneously with such alternatives. The
resulting parallelism is called or-parallelism. Thus, or-parallelism corresponds to the parallel exploration of branches in the derivation tree corresponding to different clauses which match a given goal. This kind of parallelism is specially useful when solving non-deterministic problems, i.e., problems whose solution involves a large amount of search. From the problem point of view, or-parallelism corresponds to looking for alternative solutions to the problem in parallel.

Consider now the execution of one of the alternatives, for example \texttt{crew}, in Figure 1.1. Now, to satisfy \texttt{crew}, \texttt{both navigator(peter)} and \texttt{pilot(z)} need to be satisfied. Parallelism can also be achieved if these two goals are executed in parallel. This is called and-parallelism, i.e., and-parallelism refers to the parallel execution of the goals in the body of a clause (or, more precisely, of the goals in a resolvent). And-parallelism may appear in both non-deterministic and deterministic programs, and it corresponds directly to the more "conventional" view of parallelism present in other languages, as solving simultaneously independent parts of the problem.

From the discussion above, it is clear that logic programs offer great potential for parallelism and the temptation exists to directly make use of all this potential. However, it should be noted right away that exploiting parallelism in LP can in some cases result in a slow-down, which is in conflict with the final aim of parallelism. In fact, and following [76], it is interesting to reflect on what such ultimate objective is. Our notion of parallel computation defines it as the simultaneous execution of different parts of a computation by different physical agents, and thus distinguishes it for example from multiprogramming, i.e., from the interleaved execution of different parts of a computation by a single processing agent. While for example multiprogramming can be useful for many purposes (for example, providing reasonable response times and apparent availability of the processing agent to various users of an operating system) the central usefulness of parallel execution is undoubtedly speed.\footnote{Note that, if speed is not an issue, parallelism can achieve the same effects as multiprogramming, but probably at a higher cost.} Thus, when developing a parallel execution model for a programming language, and, in our case, for logic programs, the above stated objective of achieving speed seems primordial.

Therefore, and returning to our original point above, the purpose of the parallel execution model should probably not be to achieve the maximum amount of parallelism (i.e., the maximum number of tasks available for parallel execution) but rather to achieve the maximum speed, while, of course, computing the same solution (and perhaps preserving other user "observables" such as side-effects, if they are present). In practice, "maximum speed" may be difficult to achieve or even define, but simpler concepts, such as ensuring "effective speedup" (guaranteeing that the parallel execu-
tion will be faster than the sequential one) or “no slow-down” (guaranteeing that the parallel execution will be no slower than the sequential one) can be used.

Summarizing our discussion above, we argue that there are two objectives that are highly desirable for parallel models in general, and parallel logic programming models in particular: \textit{correctness} and \textit{efficiency} \cite{75}. By correctness we refer herein to a combined notion of soundness and completeness of the parallel execution with respect to the standard sequential execution. Efficiency can be understood at two levels: one is preservation of the amount of work, which ensures speedup (modulo scheduling and communication overheads). The other, more lax, requirement is to simply ensure that the no slow-down property holds. In the rest of this document, and unless otherwise stated, we will refer to the latter when discussing efficiency.

Returning to the issue of exploiting parallelism in logic programs, the objective can then be seen as determining which of all the opportunities for parallelism available are really profitable from the correctness and efficiency points of view. It turns out that when or-parallelism is considered, and if all solutions to the problem are desired, both correctness and efficiency can be guaranteed \cite{75}. This directly results from the inherent independence among the different branches which are executed in parallel. This simplicity has contributed to the rapid development of parallel frameworks based on the or-parallel models (see, for example, \cite{152, 103, 102, 3, 144} and their references). The associated performance studies have shown good performance for non-deterministic programs in a number of practical implementations \cite{144, 3}. Furthermore, this kind of parallelism can be easily extended to CLP, as shown by the implementations of \cite{69} and \cite{115}, and, presumably, to dynamically scheduled languages.

On the other hand, guaranteeing correctness and efficiency in and-parallelism is less straightforward. Furthermore, extending and-parallelism to the context of dynamically scheduled constraint logic languages is not a trivial task.

1.2 And-Parallelism

The main problems which appear in and-parallelism are related to the fact that dependencies may exist among the goals to be executed in parallel, due to the presence of shared variables at run-time. It turns out that when these dependencies are present arbitrary exploitation of and-parallelism does not guarantee efficiency, and, if certain “impure” predicates that are relatively common in Prolog programs are used, even correctness cannot be guaranteed.

Assuming such a \textit{dependency} among the set of goals to be executed in parallel is detected, different alternatives have been proposed to solve the problem. One possible
approach is to execute the goals in parallel and in different environments, generating all solutions for each parallel goal, and later perform a join operation. This solution is correct for pure programs, but generally perceived as impractical due to the need for additional intermediate storage and the generation of much additional work. This can in fact, at the limit, affect the completeness of the approach by transforming a finite computation into an infinite one.

Another approach is to sequentialize dependent goals, as proposed by Conery [36]. Thus, only goals which are independent are allowed to execute in parallel, the rest being executed sequentially, preserving the order assumed by the sequential model. This approach is usually referred as the independent and-parallelism model. The advantage of this model is that, as shown by Hermenegildo and Rossi [75], it has been proved to fulfill both the correctness and the efficiency requirements and to be amenable to an efficient implementation if combined with compile-time analysis [54, 78, 80, 95, 99, 155, 71]. The main drawback associated with this model is that a loss of a certain amount of existing parallelism can be introduced since dependent goals are forced to execute sequentially.

Yet a different approach is to run dependent goals in parallel. The variables that are shared by the parallel goals are used as communication channels, and partial bindings of these variables are incrementally passed as streams from the producer to the consumers. This approach is usually referred to as the stream and-parallelism model. The main drawback of this model is the problems appearing in the implementation of backtracking in the presence of non-determinism. Therefore, many systems which exploit this type of parallelism give up the true non-deterministic search ("don't know" non-determinism) present in logic programming. These systems implement instead committed-choice (i.e., "don't care") non-determinism: once the system commits to a path in the execution tree, it does not explore any other paths in the tree, and if this path does not lead to a solution there is a global failure. While this approach is very interesting from the concurrency point of view, it is not as appealing for general purpose logic programming languages since “don’t-know” non-determinism is regarded as one of the most interesting and useful features of logic programming. PARLOG [24], Concurrent Prolog [139], and GHC [148] are examples of committed-choice languages.

More recently, some new and interesting approaches have been proposed. Two very related approaches are the PNU-Prolog approach of Naish [128] and the Andorra model proposed by D.H. Warren [9]. For the sake of brevity we concentrate on the latter, which is based on the basic Andorra principle. This principle states that deterministic reductions can be performed ahead of time and possibly in parallel. Thus, in this approach only deterministic goals, or, more precisely, deterministic reductions, are allowed to run
in parallel, aside from whether they are dependent or not. When no deterministic work
is available, then a non-deterministic choice is made. The advantage of this model is
that it achieves essentially the same results as the committed-choice languages, while
preserving non-deterministic search. The approach can in fact be seen as a restricted
form of the stream and-parallelism models, and it has been implemented in the “An-
dorra” family of languages [9, 135]. The main drawback is that it also misses out
on opportunities for parallelism, since it will not execute in parallel non-deterministic
reductions, even if they are independent, as the independent and-parallel model can.
Also, the low level synchronization required comes at the expense of sequential exe-
cution speed. The approach used in PNU-Prolog, in which deterministic bindings are
allowed to proceed in parallel (and goals delayed until they are deterministic) is con-
ceptually very similar, the Andorra model reflecting mostly a change of “granularity”
with respect to this model.

Another approach has been proposed by K. Shen [140] building on the work on
independent and-parallelism. The main idea is also to run independent goals in parallel.
However, certain other goals are also allowed to run in parallel, provided their shared
variables are marked as “dependent”. Such dependent goals execute in parallel but
synchronizing at a finer level through the dependent variables. The basic idea is to
solve the binding conflicts by using a priority system at the binding level – bindings
which would be executed first in the sequential model have a higher priority. Goals are
allowed to run in parallel up to the point in which they try to bind a shared variable to
a non variable term. At this point, only the goal with the highest priority is allowed to
bind the shared variable, the rest being suspended, waiting for the variable to be bound.
Note that in this model, in contrast to the one above, independent (both deterministic
and non-deterministic) goals are allowed to run in parallel, and also independent parts of
goal executions are allowed to run in parallel. The main drawback of this model, usually
referred to as the DDAS model, is the overhead represented by the management of the
dependent variables and the priority scheme, and the additional complexity introduced
in the implementation of backtracking.

There are also models which combine several of the models above. In general and-
parallelism can be combined with or-parallelism [95, 64, 135, 62, 142, 141, 63]. Much
current work is also being devoted to combining models of and-parallelism based on
determinacy and those based on independence [65, 154, 89, 76], to which we will return
later.

In this thesis we will focus on the extension to the more general context of CLP
and dynamically scheduled languages of the independent and-parallelism model. There
are several reasons for this choice. First, this model preserves the non-deterministic
search lost in many stream and-parallelism models. With respect to the models which support don’t know nondeterminism and allow (dependent) deterministic goals to be run in parallel, both more and less parallelism can be achieved when using the independent and-parallelism model and thus the choice is in principle less clear. We choose to study the independent and-parallelism model because, first, it allows us to ensure that the no slow-down property holds, which, at least in principle, is not ensured in determinacy-based models. Also, while the extension of the concept of determinacy to constraint, dynamically scheduled languages is in principle straightforward, the extension of the concept of independence is clearly more involved and deserves further attention. Finally, we believe that the emphasis on independence does not detract in any way from also considering the other models, since the extensions that we will propose for the independent and-parallelism model can be applied directly or indirectly to them. For instance, all results will be of direct use in the DDAS model since, as pointed out by Hermenegildo, this approach can also be seen as an instance of the independent and-parallelism model, but applied at a finer level of granularity, i.e., instead of requiring independence at the goal level, it is required at the binding level. Also, as pointed out by King and Soper, for the stream and-parallel models, allowing all goals to be run in parallel can in fact be inefficient. Thus, in practice, a scheduling analysis is needed to identify a partial scheduling of processes which is consistent with the program’s behavior. Such an analysis is essentially a dependence analysis similar to the one developed for detecting independence. In addition, and as mentioned before, much work is being devoted to combining models of and-parallelism based on determinacy with those based on independence, a combination which can only benefit from a further understanding of the concept of independence. Finally, there is already evidence of much cross-fertilization among models: for example, the automatic parallelization techniques by means of combining local analysis and run-time checking with sophisticated data-flow analysis that will be presented in the next section, have also proved useful in the implementation of systems in which determinacy is the parallelization principle.

\footnote{In fact, as shown in [76], the no-slowdown property can also be preserved when using determinacy as the parallelization principle, provided the principle is not also applied to the selection rule, as is done in the models previously proposed. Note, however, that this particular notion of determinacy simply as a parallelization principle and separate from the selection rule is actually contained in the independence rules that we will propose.}
1.3 Independence, Automatic Parallelization and Data-Flow Analysis

At this point it should be clear that the notion of independence is vital in parallelism. It is the main parallelization principle in both the independent-and-parallelism model and the DDAS model, and is included in the combined models recently proposed [154, 65, 89] (sometimes referred to as the concept of “stability”). The concept of independence refers to *the conditions that the run-time behavior of the goals to be run in parallel must satisfy to guarantee the correctness and efficiency of the parallelization with respect to the sequential execution.*

If the parallelization is performed manually by the programmer, by means of a language that includes parallel constructs, his/her knowledge about the behavior of the program goals will be the basis for deciding if the goals satisfy the independence conditions and thus can be run in parallel. However, this adds an additional dimension of complication to the already complicated and bug-prone task of programming. The best programming environment would appear to be one in which the programmer can freely choose between only concentrating on the conventional programming task itself (letting a parallelizing compiler uncover the parallelism in the resulting program) or, alternatively, performing, in addition, the task of explicitly annotating parts of the program for parallel execution. In the latter case, the compiler should be able to aid the programmer in this task. Ideally, different choices should be allowed for different parts of the program. In any case, some degree of automatic parallelization seems desirable.

It follows from our discussion that in and-parallelism, automatic parallelization is related closely to the detection of some notion of independence. Therefore, the dependencies among the different goals must be determined, and there is a related parallelization overhead involved. It is vital that such overhead remain reasonable. In the first models [36], such dependencies were in principle checked at run-time. It turns out that such run-time checking can result in substantial overhead, related to having to consider all possible combinations of goals in all resolvents for parallelization, and also checking for the independence of each such combination. Furthermore, since independence in general cannot be detected at run-time without executing the goals (it is defined in terms of the run-time behavior of the goals), the run-time tests are based on quite restrictive sufficient conditions. While such conditions do guarantee the independence of the goals, they may miss opportunities for parallelization. We refer to this as *a priori* detection of independence.

As an alternative, Chang et al [22], proposed performing independence detection
exclusively at compile-time, by using data-flow analysis techniques, and thus avoiding any run-time support. This approach had the unquestionable merit of pointing in a very promising direction, but, as proposed, it had the drawback of being too restrictive, and thus often being unable to detect much available parallelism in programs. An interesting way of reducing such overheads, initially proposed by DeGroot [54] and further developed by Hermenegildo [78, 75], is to mark at compile-time selected literals in the program for parallel execution and, when independence cannot be determined statically, generate a reduced set of efficient parallelization tests, to be checked at run-time. An alternative, proposed by Lin [100], was to avoid choosing a particular expression at compile-time and performing a dynamic sequencing of the goals based on a bit-vector encoded dependency graph.

The last approaches have the advantage of reducing the independence checking overhead with respect to fully dynamic models, while achieving more parallelism than fully static models. However, they still suffer from the drawback that the conditions that are generated are often too costly to compute at run-time. This is due to the fact that at the time the above proposals were made, global data-flow analysis for logic programs was in its infancy and thus the parallelization usually made use only of local analysis – i.e., analysis of each program clause considered in isolation instead of as part of a program. Furthermore, since these kinds of analyses could not infer accurate information regarding the run-time behavior of the goals, compile-time independence detection was still based on the sufficient conditions which enable an a priori detection.

A more effective approach, proposed initially by R. Warren et al [155, 77] and developed further in [123, 19] is to combine local analysis and run-time checking with a highly sophisticated data-flow analysis based on the technique of abstract interpretation [38]. This technique, initially viewed as only a theoretical exercise, was after some time applied to logic programming [117, 50, 143], and shown to be quite useful in practice [155, 77]. Since then, different abstract interpretation-based analyzers have been defined for the independent and-parallelization task [26, 84, 125, 124] and some analyses proposed in other contexts have also been found useful [143, 91, 50, 52, 53, 112]. However, these preliminary studies must be completed by performing an exhaustive study of the importance of state of the art data-flow analysis tools in the automatic parallelization task.

The introduction of sophisticated compile-time analysis in the parallelization process has suggested that the need for a strict “a priori” condition can be somewhat relaxed, since the analysis can to some extent predict the behavior of the goals at run time [74, 75, 19] (of course, the whole process is still “a priori,” but we understand it to be less strict because of the need of a prediction of goal behavior). The task of
such an analysis is to infer accurate information regarding the bindings of variables at run-time, so that independence of the goals, in the most general sense, can be guaranteed. When this kind of independence cannot be detected, the information will be used to simplify the tests as much as possible, based on a priori conditions, generated for run-time evaluation.

1.4 Research Goals

This thesis builds on most of the work presented in the previous sections with the final aim of extending the applicability of parallelism based on independence, as presented in the work of Hermenegildo and Rossi for logic programs, to the more general class of constraint logic programs with dynamic scheduling. In doing this, our aim is also to further study the effectiveness of traditional global analysis and parallelization techniques and extend such techniques to the new type of programs being considered.

We will first extend the independence concept and the a priori conditions developed for LP to the more general context of CLP and dynamically scheduled languages. Furthermore, we will show that generalizing independence to arbitrary CLP languages and constraint solvers yields new insights into independence. Then we will perform an exhaustive evaluation of global data-flow analysis techniques in the automatic parallelization task. The results obtained from this experimental evaluation show that the effectiveness of the parallelization significantly depends on the information provided by global data-flow analysis. Therefore, such analysis must be available in the context of dynamically scheduled constraint logic languages.

Regarding CLP languages, a few general frameworks have already been defined for this purpose [108, 32, 12]. However, one common characteristic of these frameworks is that they are either not implementation oriented or they depart from the approaches that have been so far quite successful in the analysis of traditional logic programming languages. We will show how some of the techniques which have been used with great success in LP, and for which efficient fixpoint algorithms have already been developed, can relatively easily be extended to the analysis of CLP programs. Regarding languages with dynamic scheduling, it turns out that the global data-flow analyses used in the compilation of traditional logic programs are not correct in this context. We will also develop a framework for global data-flow analysis of logic languages with dynamic scheduling.
1.5 Thesis Overview

The thesis is divided into two parts. Part I addresses the extension of the independence concept to the context of dynamically scheduled constraint logic languages. Part II is devoted to the important enabling techniques of abstract interpretation based analysis.

After introducing the notation and some background material in Chapter 2, Part I starts with Chapter 3 where we extend the concept of independence to the more general context of constraint logic programming, give sufficient conditions that are easier to detect at run-time than the definition of independence, and prove the correctness and efficiency of the approach. We argue that generalizing independence to arbitrary CLP languages and constraint solvers yields new insights into independence. In particular, independence has traditionally been expressed in terms of search space preservation but this relationship has never been formalized. We show that search space preservation is, in the context of LP languages, not only a sufficient but also a necessary condition for ensuring correctness and efficiency. Furthermore, such formalization shows that the generalized conditions for search space preservation are necessary but no longer sufficient for ensuring the efficiency of several optimizations when arbitrary CLP languages are taken into account. The reason is that even if search space is preserved, the cost of executing a set of primitive constraints may depend on the order in which those primitive constraints are considered. Thus, optimizations which vary the intended execution order established by the programmer, such as parallel execution, can actually cause execution to slow down. In order to ensure efficiency, we must therefore consider an additional issue – “independence of constraint solving” – which characterizes the properties of the constraint solver behavior when changing the order in which constraints are added. This issue has not risen previously because the standard unification algorithm is independent in this sense for many practical programs and for the usual implementation of the algorithm. However, in the more general context of CLP, constraint solver independence need not hold.

Chapter 4 performs a further generalization of the notion of independence in constraint logic programs to the context of constraint logic programs with dynamic scheduling. We show that the traditional concepts of independence are not useful in this context and propose new concepts of independence for constraint logic languages with dynamic scheduling. We propose several a priori sufficient conditions for independence and also give correctness and efficiency results for parallel execution of constraint logic programs based on the proposed notions of independence.

Having defined the notions of independence needed and knowing that global data-flow analysis has proven to be a key technology in program parallelization, Part II of the dissertation is devoted to studying global data-flow analysis for program parallelization.
and generalizing it to deal with the languages under consideration.

In Chapter 5 we perform an exhaustive study aimed at assessing the effectiveness of data-flow analysis tools in the task of compile-time automatic parallelization of logic programs, based on strict independence detection. Five analyzers are studied: the ASub domain of Sondergaard [143] with the abstract functions presented in [26]; the Sharing domain of Jacobs and Langen [84] with the abstract functions presented in [125]; the Sharing+Freeness abstract domain and abstract functions defined in [124]; and the analyzers resulting from the combination of Sharing and ASub, and Sharing+Freeness and ASub based on the reduced product approach of Cousot and Cousot [39], further developed in [30]. Then each analyzer is evaluated in terms of its accuracy, i.e., its ability to determine the actual dependencies among the program variables, its efficiency, i.e., the time taken by the analyzers for determining such dependencies, and its effectiveness, measured in terms of the ultimate performance of the parallelized programs, i.e., the speedup obtained with respect to the sequential version. Our overall conclusion is that global analysis based on abstract interpretation is indeed an effective and practical tool in this application.

Chapter 6 deals with the issue of global data-flow analysis of constraint logic programs. We show how some of the global analysis techniques which have been used with great success and practicality in LP, and for which efficient fixpoint algorithms have already been developed, can relatively easily be extended to the analysis of CLP programs. The generalization proposed should be quite useful even beyond parallelization, since the associated optimizations performed in the context of LP (other than parallelization) also appear equally applicable to the context of constraints. Indeed, the high cost of performing constraint satisfaction makes the potential performance improvements even larger and suggests opportunities for further optimizations.

In the chapter we illustrate the point above by proposing a simple but quite general and powerful extension of Bruynooge’s traditional framework [10] which makes it applicable to the analysis of CLP programs. We also extend the framework to deal with non-normalized programs and with passive constraints. Finally, we give correctness conditions for the resulting framework. The objective is thus to fully specify a generic algorithm for analysis of CLP programs. Then, using this generalized framework, we propose a simple abstract domain, ConsP, and its required abstract functions. This abstract domain is aimed at approximating definiteness information by determining whether program variables are definite, i.e., constrained to a unique value. The domain keeps track in a compact way of sure dependencies among variables in order to accurately propagate definiteness information. This analyzer has been implemented within the abstract interpretation system PLAI [121, 122, 16]. Results from this implementa-
tion showing its efficiency and accuracy are also presented.

In Chapter 7 we develop a framework for global data-flow analysis of logic languages with dynamic scheduling. This provides the basis for optimizations which remove the overhead of dynamic scheduling and promises to make the performance of logic languages with dynamic scheduling competitive with that of traditional Prolog. This includes the automatic parallelization of such languages within the independent and-parallelism model.

In doing this we first give a denotational semantics for languages with dynamic scheduling. This provides the semantic basis for our generic analysis. Second, we give a generic global data-flow analysis algorithm which is based on the denotational semantics. Correctness is formalized in terms of abstract interpretation. The analysis gives information about call arguments and the delayed calls, as well as implicit information about possible call schedulings at runtime. Finally, we demonstrate the utility and practical importance of the data-flow analysis algorithm. We sketch an example instantiation of the generic analysis which gives information about groundness and freeness of variables in the delayed and actual calls. Information from the example analysis can be used to optimize target code in many different ways. In particular, it can be used to reduce the overhead of dynamic scheduling by removing unnecessary tests for delaying and awakening and by reordering goals so that atoms are not delayed. It can also be used to perform optimizations used in the compilation of traditional Prolog such as: recognizing determinate code and so allowing unnecessary backtrack points to be deleted; improving the code generated for unification; recognizing calls which are “independent” and so allow the program to be run in parallel, etc. Preliminary test results, given in the chapter, show that the analysis and associated optimizations used to reduce the overhead of dynamic scheduling give significant improvements in the performance of these languages.

Finally, Chapter 8 offers some conclusions and directions for future work.

1.6 Main Contributions

We now enumerate the main contributions of the thesis. Some of these contributions have been previously published and developed in the context of collaborative work with researchers other than the thesis supervisor. The relevant publications and relationships are also specified.

- Generalizing the notion of independence to the context of constraint logic programs. Showing that the traditional concepts of independence are not useful in this context and proposing concepts of independence for constraint logic programs
(weak, strong, and search independence). Proposing several a priori sufficient conditions for independence. Introducing the notion of solver independence and justifying the need for it. Showing also the applicability of the concepts in other applications such as reordering and intelligent backtracking. (Part of this work was done in collaboration with Kim Marriott, of IBM T.J. Watson Research Labs and Monash University and a summary was published in the 1993 International Logic Programming Symposium [46].)

- Showing correctness and efficiency results for parallel execution of constraint logic programs based on the proposed notions of independence. Establishing a relationship between the new concepts of independence and the traditional ones for logic programs. As a result, clarifying the relationship between independence and search space preservation in logic programs and also offering the most general form of (non-strict) independence proposed so far for traditional logic programming.

- Generalizing the notion of independence in constraint logic programs to the context of constraint logic programs with dynamic scheduling. Showing that the traditional concepts of independence are not useful in this context and proposing new concepts of independence for such context. Proposing several a priori sufficient conditions for independence. Showing correctness and efficiency results for parallel execution of constraint logic programs based on the notions of independence proposed. (Part of this work was done in collaboration with Kim Marriott, of IBM T.J. Watson Research Labs and Monash University.)

- Performing an exhaustive study of the effectiveness of global analysis in strict independence-based automatic parallelization of logic programs, including an implementation of several domains and substantial improvement of an existing analyzer. This represents the first exhaustive study of the performance of global analysis in parallelization in terms of speedups. (Part of this work was done in collaboration with Francisco Bueno, of the Technical University of Madrid (UPM) with the help of other members of the CLIP group. The combination of the domains based on the reduced product approach was done in collaboration with Michael Codish, Anne Mulkers, and Maurice Bruynooghe of the Catholic University of Leuven and a summarized version published in the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation [30].)

- Applying these techniques to the domains used in the previously mentioned effectiveness study and extending such study to include the combined domains. 15
• Showing that conventional generic global analysis frameworks are still largely useful for analyzing constraint logic programs while the abstract domains need to be revisited. Proposing modifications to such frameworks and proposing and implementing a generalization of Bruynooghe’s framework capable of analyzing constraint logic programs. Proposing and implementing a simple definiteness analysis. Studying the effectiveness of the framework using this analysis. (This work is an extension of the work published in the 1993 International Logic Programming Symposium [44], and has benefited from collaboration with Veroniek Dumortier, Maurice Bruynooghe, and Gerda Janssens of the Catholic University of Leuven.)

• Proposing a framework for the analysis of constraint logic programs with dynamic scheduling. Developing a denotational semantics for such programs and an abstraction based on this semantics. Implementing a complete global analyzer for constraint logic programs with delay. Studying its performance in the optimization of programs which contain delay declarations. Applying the results to the parallelization of constraint logic programs with dynamic scheduling. (Part of this work was done in collaboration with Kim Marriott, of IBM T. J. Watson Research Labs and Monash University and a summary was published in the 1994 ACM Conference on Principles of Programming Languages POPL [106].)
Chapter 2

Background

In this chapter we present the notation and basic notions which will be used throughout the thesis. We also provide a brief introduction to the logic programming paradigm and to the abstract interpretation theory.

2.1 Basic Definitions

Lattices

An equivalence relation on a set $A$ is a binary relation $R$ on $A$ which is reflexive, symmetric, and transitive. Let $\approx$ be an equivalence relation on a set $A$. Equivalence classes in $A$ with respect to $\approx$ will be denoted by $[a]_\approx$ (i.e., $[a]_\approx = \{a' \mid a \approx a'\}$). $\equiv$ denotes identity for any syntactic object.

A preorder is a binary relation $R$ which is reflexive and transitive. A partial order is a preorder which is also antisymmetric. A poset $P_\sqsubseteq$ is a set $P$ equipped with a partial order $\sqsubseteq$. Given a poset $P_\sqsubseteq$ and $X \subseteq P$, $y \in P$ is an upper bound for $X$ iff for each $x \in X : x \sqsubseteq y$. An upper bound $y$ for $X$ is the least upper bound (often denoted by lub$_\sqsubseteq$) iff for every upper bound $y' : y \sqsubseteq y'$. Lower bounds and greatest lower bounds (often denoted by glb$_\sqsubseteq$) are defined dually. A chain is a (possibly empty) subset $X$ of $P$ such that for all $x, x' \in X : x \sqsubseteq x'$ or $x' \sqsubseteq x$. A directed set is a poset in which any two elements, and thus any finite subset, has an upper bound in the set. A complete partial order (denoted by cpo) is a poset such that every increasing chain has a least upper bound.

A lattice is a poset $P_\sqsubseteq$ in which any two elements have glb and lub. A complete lattice is a poset $L_\sqsubseteq$ for which all subsets have lub and glb. Any finite lattice is complete. A lattice $L$ with partial ordering $\sqsubseteq$, least upper bound $\sqcup$, greatest lower bound $\sqcap$, infimum $\bot$ ($\bot = \sqcup\emptyset = \sqcap L$), and top element $\top$ ($\bot = \sqcap\emptyset = \sqcup L$) is denoted by $L_\sqsubseteq(\bot, \top, \sqcup, \sqcap)$.
When clear from the context, we use the poset notation to denote lattices. From the point of view of data-flow analysis, we will often make use of special types of lattices. In particular, we will consider Noetherian lattices, namely lattices where all strictly increasing chains are finite (often called the ascending chain condition).

Fixpoint Theory

We write \( f : A \to B \) to mean that \( f \) is a total function of \( A \) (the domain) into \( B \). Let \( f : A \to B \) be a mapping, for each \( C \subseteq A \) we denote by \( f(C) \) the image of \( C \) by \( f \), i.e., \( f(C) = \{ f(x) \mid x \in C \} \). The domain of a function \( f \) is often denoted by \( \text{dom}(f) \). The range of a function is \( \text{range}(f) = f(\text{dom}(f)) \). Functions from a set to the same set are called operators. The set of all operators \( (L \to L) \) on the complete lattice \( L \) is a complete lattice.

Let \( A = (\bot, \top, \sqcup, \sqcap) \) and \( B = (\bot, \top, \sqcup, \sqcap) \) be complete lattices. A function \( f : A \to B \) is monotonic iff for all \( x, x' \in A : x \sqsubseteq A x' \Rightarrow f(x) \sqsubseteq B f(x') \). \( f \) is continuous iff for each non-empty chain \( X \subseteq A : f(\sqcup A X) = \sqcup B f(X) \). A function \( f \) is additive iff the previous condition is satisfied for each non-empty set \( X \subseteq A \).

Let \( f \) be a monotonic operator on \( L = (\bot, \top, \sqcup, \sqcap) \). The set of fixpoints of \( f \) is \( \text{fp}(f) = \{ x \mid x = f(x) \} \). The fundamental theorem of Tarski states that the set of fixpoints \( \text{fp}(f) \) of a monotonic function \( f \) is a complete lattice. The least fixpoint of a function \( f \) is denoted \( \text{lp}(f) \). Dually, there is a greatest fixpoint denoted \( \text{gp}(f) \).

2.2 Logic Programming

In this section we introduce the basic concepts and notation that will be used throughout this thesis when dealing with logic programs\(^1\). We follow mainly [5, 101].

Syntactic Objects

Upper case letters generally denote collections of objects, while lower case letters generally denote individual objects. \( u, v, w, x, y, z \) will denote variables, \( t \) will denote a term, \( p, q \) will denote predicate symbols, \( f, g \) will denote function symbols, \( a, h \) will denote atoms, \( \theta, \sigma \) will denote substitutions, \( \rho \) will denote a renaming, \( b, l \) will denote literals, \( P, Q \) will denote programs, and \( R \) will denote a relation.

Let \( \Sigma, \Pi \) and \( \text{Var} \) denote a set of function symbols, a set of predicate symbols and a denumerable set of variables, respectively. A predicate or function symbol \( f \) with arity \( n \) is written \( f/n \).

\(^1\)We will only deal with positive logic programs.
We denote by $T(\Sigma, \text{Var})$ (or $\text{Term}$, when clear from the context) the set of terms constructed from $\Sigma$ and $\text{Var}$. Likewise, we denote by $A(\Pi, \Sigma, \text{Var})$ (or $\text{Atom}$) the set of atoms of the form $p(t_1, \cdots, t_n)$, where $p \in \Pi$ and $t_i \in T(\Sigma, \text{Var})$ ($1 \leq i \leq n$). Together $\Sigma, \text{Var},$ and $\Pi$ define a \textit{first-order language}.

A \textit{goal} is a sequence of atoms, generally denoted by $g$ or $G$, possibly subscripted. Note that although a goal is a sequence of atoms it will be denoted by a lowercase letter. This has been done for reasons of readability due to the heavy use we will make of goals. Furthermore, although concatenation of sequences $A$ and $B$ will be denoted by $A::B$, in the case of goals we will use $g_1 : g_2$. The empty atom sequence is denoted by $\text{nil}$, and often omitted. A \textit{(definite) clause} is an object of the form $h \leftarrow b_1, \cdots, b_n$ with $n \geq 0$, where $h$ is an atom called the \textit{head}, $b_1, \cdots, b_n$ is a sequence of atoms called the \textit{body}. The set of clauses constructed from elements of $\text{Atom}$ is denoted $\text{Clause}$. A \textit{unit clause} (or \textit{fact}) is a clause with an empty body. A \textit{(definite) logic program} is a set of clauses.

We let $\text{vars}(t)$ denote the set of variables occurring in a term $t$. A term $t$ is \textit{ground} if $\text{vars}(t) = \emptyset$. These notions can be extended to any syntactic object.

**Substitutions**

As \textit{substitution} is a (finite) mapping from $\text{Var}$ to $\text{Term}$. Given a substitution $\theta$, $\text{dom}(\theta)$ denotes its domain and $\text{range}(\theta)$ denotes the set of variables which appear in a term from the range of $\theta$. We will represent $\theta$ by $\{x/t \mid x \in \text{dom}(\theta), \theta(x) = t\}$. A pair $x/t$ is called a \textit{binding}. We assume that for each binding $x/t$ in a substitution, $x \not\equiv t$. The empty substitution is denoted $\epsilon$. The application of a substitution $\theta$ to a syntactic object $s$ is denoted by $s\theta$ and it is defined as the syntactic object obtained by replacing each variable $x$ in $s$ by $\theta(x)$. If $\Theta$ is a set of substitutions, then $S\Theta = \{S\theta \mid \theta \in \Theta\}$.

A variable $x$ is \textit{ground} with respect to a substitution $\theta$ if $\theta(x)$ is ground. A variable $x$ is \textit{linear} with respect to a substitution $\theta$ if $\theta(x)$ contains only single occurrences of variables. A set of variables $\{x_1, \cdots, x_n\}$ are \textit{aliased} or \textit{share} with respect to a substitution $\theta$ if $\text{vars}(\theta(x_1)) \cap \cdots \cap \text{vars}(\theta(x_n)) \neq \emptyset$. $\theta_s$ denotes the restriction of the substitution $\theta$ to the variables occurring in $s$, which is extended as an identity for each variable $x \in \text{var}(s)$ such that $\theta(x)$ is undefined.

Composition of substitutions $\theta$ and $\sigma$ is defined as function composition and denoted $\theta \sigma$. A \textit{renaming} is a substitution $\rho$ for which there exists the inverse $\rho^{-1}$, that is, $\rho \rho^{-1} = \rho^{-1} \rho = \epsilon$. A substitution $\theta'$ is \textit{more general} than $\theta$, represented by $\theta \leq \theta'$, iff there exists $\sigma$ such that $\theta = \theta' \sigma$. A substitution $\theta$ is \textit{idempotent} if $\theta \theta = \theta$. We are only interested in idempotent substitution. The set of idempotent substitutions is denoted $\text{Sub}$.
An equation is an entity of the form \( t = u \) where \( t \) and \( u \) are terms. A substitution \( \theta \) is a unifier of a set of equations \( E \) iff for all \( (t = u) \in E \), \( t\theta = u\theta \). If such a unifier exists, \( E \) is said to be unifiable. Let \( \text{Unif}(E) \) be the (possibly empty) set of unifiers of \( E \). A substitution \( \theta \) is a most general unifier of \( E \) iff \( \theta \) is more general than any other unifier of \( E \). If \( E \) has a most general unifier, it has an idempotent most general unifier.

The set of most general unifiers of \( E \) is denoted by \( \text{mgu}(E) \). The function \( \text{mgu} \) maps a pair of syntactic objects to an idempotent most general unifier of the objects, if it exists. Otherwise it returns \text{Fail}.

There is a natural bijection between substitutions and sets of equations in solved form. We let \( \text{eqn}(\theta) \) denote the set of equations corresponding to the substitution \( \theta \).

A term \( t' \) is more general (or less instantiated) than a term \( t \), represented by \( t \leq t' \), iff there exists \( \theta \) such that \( t = t'\theta \). This notion can be extended to any syntactic object. Syntactic objects \( t_1 \) and \( t_2 \) are equivalent up to renaming, denoted by \( t_1 \sim t_2 \), iff there exists a renaming \( \rho \) such that \( t_1\rho = t_2 \). Term/ \sim , Atom/ \sim and Clause/ \sim are complete lattices with respect to \( \leq \).

Given an equivalence class (induced by renaming) \( \hat{t} \) of syntactic objects and a finite set of variables \( V \subseteq \text{Var} \), it is always possible to find a representative element of \( \hat{t} \) which contains no variables from \( V \). For a syntactic object \( t \) and a set of equivalence classes (by renaming) of objects \( I \), we denote by \( \langle c_1, \cdots, c_n \rangle \ll t I \) that \( c_1, \cdots, c_n \) are representative elements of \( I \) renamed apart from \( t \) and from each other.

**Declarative Semantics**

Given a first order language \( L \), an interpretation \( I \) for \( L \) consists of:

- a non empty set \( D \) called the domain of interpretation
- an assignment for each constant in \( L \) of an element in \( D \)
- an assignment for each \( n \)-ary function in \( L \) of a mapping from \( D^n \) to \( D \)
- an assignment for each \( n \)-ary predicate in \( L \) of a subset of \( D^n \)

A variable assignment (or valuation) on \( D \) is a function mapping variables onto elements of \( D \). The semantics of a formula can be defined inductively on its structure. A formula \( F \) is true in a variable assignment \( v \) over \( I \) (denoted \( I \models_v F \)) iff:

- if \( F = p(t_1, \cdots, t_n) \) and \( p_I \) is the relation associated with \( p \) in \( I \):
  \[
  I \models_v p(t_1, \cdots, t_n) \iff v(t_1), \cdots, v(t_n) \in p_I
  \]

\footnote{Note that all most general unifiers of a set \( E \) are equivalent.}
\[ I \models_v true \text{ and not } I \models_v false \]

- if \( F \) and \( G \) are formulas
  \[
  I \models_v \neg F \text{ iff } I \models_v F
  \]
  \[
  I \models_v F \lor G \text{ iff } I \models_v F \text{ or } I \models_v G
  \]
  \[
  I \models_v \forall x F \text{ iff } I \models_{v[x/d]} F \quad \text{for all } d \in D
  \]

The semantics for the other connectives can be defined in terms of \( \lor, \forall \) and \( \neg \). A formula is definitely true in an interpretation \( I \) iff it is true for each variable assignment. We say that an interpretation \( I \) is a model for a set of formulas \( S \) iff every formula in \( S \) is true in \( I \). When \( S \) has a model, we say that \( S \) is satisfiable or consistent. Given two sets of formulas \( S \) and \( S' \), we say that \( S' \) is a semantic consequence of \( S \) iff every model of \( S \) is also a model of \( S' \). If \( S' = \{ F \} \) then \( F \) is a logical consequence of \( S \).

Let \( \Sigma, \text{Var} \) and \( \Pi \) be a fixed first-order language \( L \). By the Herbrand universe we mean the set of ground terms \( T(\Sigma, \emptyset) \). By the Herbrand base we mean the set of all ground atoms \( A(\Pi, \Sigma, \emptyset) \). If \( L \) is associated with a program \( P \), we denote the Herbrand universe and base as \( U_P \) and \( B_P \) respectively.

By an Herbrand interpretation we mean an interpretation with domain \( U_P \), which assigns each constant or function in \( L \) with the corresponding constant and function in \( U_P \) and, for each \( n \)-ary relation in \( L \), an element in \( \wp(\wp(U_P)^n) \). Thus, an Herbrand interpretation is uniquely determined by a subset of \( B_P \), Herbrand models are defined as usual. If a set of formulas \( S \) (in clausal form) has a model, then it has an Herbrand model.

Let \( P \) be a program and \( \{ M_i \}_{i \in \mathbb{I}} \) be the non-empty set of all Herbrand models. Then \( \bigcap_{i \in \mathbb{I}} M_i \) is a Herbrand model for \( P \) and it is called the least Herbrand model for \( P \), denoted \( M_P \). Herbrand interpretations which are models for a logic program \( P \) can be characterized in terms of the immediate consequence operator \( T_P \) associated with \( P \). \( T_P \) maps Herbrand interpretations to Herbrand interpretations:

\[
T_P(I) = \{ h \in B_P \mid h \leftarrow b_1, \ldots, b_n \in Gr(P), \{ b_1, \ldots, b_n \} \subseteq I \}
\]

where \( Gr \) is a function mapping any syntactic object to the set of its ground instances. The mapping \( T_P \) is continuous on the complete lattice of Herbrand interpretations \( \wp(B_P) \subseteq \wp \). An interpretation \( I \) is a model for \( P \) iff \( T_P(I) \subseteq P^* \). As a result, it can be concluded that \( M_P = \text{fp}(T_P) = (T_P)^w(\emptyset) \).

\[ ^3_v[x/d] \text{ is the assignment that differs from } v \text{ only for the assignment of } x \text{ to } d. \]
Operational Semantics

Logic Programs are evaluated through a combination of two mechanisms—replacement and unification. This strategy is named $SLD-resolution$. The operational semantics can be presented as a transition on states $\langle G, \theta \rangle$, where $G$ is a goal, and $\theta$ is a substitution.

The transitions in the transition system are:

- $s : (\ldots : a_{i-1} : a_i : a_{i+1} : \ldots), \theta) \rightarrow fail$ if $a_i$ is an atom selected by the computation rule and for every clause $h \leftarrow B$ of $P$ renamed to new variables, $\text{mgu}(a_i\theta, h) = fail$.

- $s : (\ldots : a_{i-1} : a_i : a_{i+1} : \ldots), \theta) \rightarrow s' : (\ldots : a_{i-1} : B : a_{i+1} : \ldots), \theta\theta'$ if $a_i$ is an atom selected by the computation rule, $h \leftarrow B$ is a clause of $P$ selected by the search rule and renamed to new variables, and $\text{mgu}(a_i\theta, h) = \theta'$.

A derivation of a state $s$ (called the initial state of the derivation) for a program $P$ is a finite or infinite sequence of transitions $s_0 \rightarrow s_1 \rightarrow \cdots$, in which $s_0 \equiv s$. A state from which no transition can be performed is a final state. A derivation is successful (also referred as a refutation) when it is finite and the final state has the form $\langle \text{nil}, \theta \rangle$. A derivation is failed when it is finite and the final state is fail. The substitution $\sigma$ is said to be a partial answer to state $s \equiv \langle G, \theta \rangle$ if there is a derivation from $s$ to a state with substitution $\sigma$. An answer to state $s$ is the last partial answer of a successful derivation.

The maximal derivations of a state can be organized into a derivation tree in which the root of the tree is the start state and the children of a node are the states the node can reduce to. The derivation tree represents the search space for finding all answers to a state and is unique up to variable renaming. Each branch of the derivation tree of state $s$ is a derivation of $s$. Branches corresponding to successful derivations are called success branches, branches corresponding to infinite derivations are called infinite branches, and branches corresponding to failed derivations are called failure branches.

The soundness of the SLD-resolution states that, if there exists a successful derivation for state $\langle G, \epsilon \rangle$ with final state $\langle \text{nil}, \theta \rangle$, then $G\theta$ is a logical consequence of $P$. Thus, if there exists a successful derivation for $\langle G, \epsilon \rangle$ in $P$, then $P \cup \{G\}$ is inconsistent. The completeness of the SLD-resolution states that if $P \cup \{G\}$ is inconsistent, there exists a successful derivation for $\langle G, \epsilon \rangle$ in $P$.

Let $\text{Succ}_P$ (success set) be the set of all ground atoms $b$ such that there is a successful derivation for $\langle b, \epsilon \rangle$ in $P$. Then, $\text{Succ}_P$ corresponds to the least Herbrand model of $P$. 

22
2.3 Abstract Interpretation

Data-flow analysis is the process of statically – at compile-time – inferring information about the properties of the variables and data structures in a program. The intended purpose of this process is to use such knowledge to improve significantly the task performed by compilers, program transformation tools, etc. The idea of approximating program properties by evaluating a program on a simpler domain of descriptions of “concrete” program states comes from the early 70’s. The basic idea is to approximate program properties from the concrete semantics into an abstract semantics equipped with a structure (e.g. ordering) which is somehow present in the richer concrete structure associated with program execution. Thus, the approximation implicitly defines a relation between the concrete and the abstract semantics of the program.

The approach of Cousot and Cousot [38] is fundamental, due to its ability to formalize this relation in a highly general setting: abstract interpretation. The connection between the concrete and abstract interpretation is based on a pair of functions – the abstraction ($\alpha$) and concretization ($\gamma$) functions – and formalized in terms of Galois connections. In this section we introduce the basic concepts of abstract interpretation as first developed in [38], in terms of Galois connections and Galois insertions.

Consider a partial order which specifies the degree of precision of any element in a poset, and an approximation function $\alpha$ which is a mapping associating to each element in $\text{Dom}_C$ an abstract object in $\text{Dom}_A$. The idea is that if $x^A = \alpha(x^C)$ and $x^A \subseteq^A y^A$, then $y^A$ is also a correct approximation of $x^C$, although less precise. This condition can be expressed as $\alpha(x^C) \subseteq^A y^A$. Dually, if $y^C = \gamma(y^A)$ and $x^C \subseteq^C y^C$, then $y^A$ is also a correct approximation of $x^C$, although $x^C$ provides more accurate information than $y^C$. This condition can be expressed by $x^C \subseteq^C \gamma(y^A)$.

**Definition 2.3.1** Galois connection

A Galois connection is a quadruple $(\text{Dom}_C, \alpha, \text{Dom}_A, \gamma)$ where:

1. $\text{Dom}_C$ and $\text{Dom}_A$ are posets called concrete and abstract domains respectively;
2. $\alpha : \text{Dom}_C \to \text{Dom}_A$ and $\gamma : \text{Dom}_A \to \text{Dom}_C$ are functions called abstraction and concretization functions respectively, satisfying that for every $d^A \in \text{Dom}_A$ and $d^C \in \text{Dom}_C$: $\alpha(d^C) \leq^A d^A \iff d^C \leq^C \gamma(d^A)^4$.

Note that for a Galois connection only one of $\{ \alpha, \gamma \}$ need be specified since (if one exists) it uniquely determines the other.

---

$^4$Initially [38] $\alpha$ and $\gamma$ were required to be monotonic. However, as pointed out in [42, 41], this requirement is redundant.
**Definition 2.3.2** Galois insertion

A Galois insertion is a Galois connection satisfying \( \alpha(\gamma(d^A)) = d^A \). ■

In addition to the properties of a Galois connection, a Galois insertion has no superfluous elements in the abstract domain, i.e., different elements of the abstract domain have different meanings. However, a Galois insertion can always be forced by considering a more concise abstract domain which is isomorphic to the quotient set \( \text{Dom}^A / \approx \), such that for each \( x^A, y^A \in \text{Dom}^A \), \( x^A \approx y^A \) iff \( \gamma(x^A) = \gamma(y^A) \).

The following formally specifies the notion of approximation (in terms of \( \gamma \)) which is then lifted from the primitive domains to function domains:

**Definition 2.3.3** approximation

Let \((\text{Dom}^C, \alpha, \text{Dom}^A, \gamma)\) be a Galois insertion and let \( \mu^C : \text{Dom}^C \to \text{Dom}^C \) and \( \mu^A : \text{Dom}^A \to \text{Dom}^A \) be monotonic functions. We say that \( d^A \in \text{Dom}^A \) \( \gamma \)-approximates \( d^C \in \text{Dom}^C \), denoted \( d^A \gg \gamma d^C \), if \( d^C \leq \gamma(d^A) \). We say that \( d^A \gamma \)-approximates \( d^C \), denoted \( d^A \gg d^C \), if for every \( d^A \in \text{Dom}^A \), \( d^C \in \text{Dom}^C \) such that \( d^A \gg d^C \) we have \( \mu^A(d^A) \gg \mu^C(d^C) \). ■

A concrete semantics is typically defined as least fixed point of an operator on programs. Typically, the meaning of a program \( P \) may be expressed as \([P] = \text{fix}(f_P)\) where \( f_P : \text{Den} \to \text{Den} \) is a monotonic operator on a domain of denotations \( \text{Den} \).

A program analysis will typically be defined by introducing an appropriate Galois insertion \((\text{Den}^C, \alpha, \text{Den}^A, \gamma)\) and constructing an approximation \( f_P^A : \text{Den}^A \to \text{Den}^A \) of \( f_P \) so that the least fixed point of \( f_P^A \) is finitely computable. This construction often takes a systematic approach which involves replacing the basic operations in the concrete semantic operator \( f_P \) by corresponding abstract operations in \( f_P^A \) (e.g., [41, 129]). Given that these abstract operations approximate the concrete operations it is generally straightforward to prove that the derived abstract semantic operator approximates the concrete semantic operator. The fundamental theorem of abstract interpretation provides the following result:

**Theorem 2.3.4**

Let \((\text{Dom}^C, \alpha, \text{Dom}^A, \gamma)\) be a Galois insertion and let \( \mu^C : \text{Dom}^C \to \text{Dom}^C \) and \( \mu^A : \text{Dom}^A \to \text{Dom}^A \) be monotonic functions such that \( \mu^A \gamma \)-approximates \( \mu^C \). Then \( \text{fix}(\mu^A) \gg \text{fix}(\mu^C) \). ■

Therefore, the “art” of abstract interpretation can be described as involving the following steps: (1) to choose an appropriate concrete semantics; (2) to identify a
suitable notion of data-description; and (3) to provide good approximations of the basic operations in the concrete semantics. Let us illustrate this point with a simple example taken from [39].

**Example 2.3.1**

Let $\text{Parity} = \{ \bot, \od, \ev, \top \}$ and $\text{Sign} = \{ \bot, 0, \dot, \ddot, \top \}$ be complete lattices ordered as illustrated in Figure 2.1. Let

\[
\begin{align*}
\gamma_{\text{parity}} &= \left\{ \begin{array}{c}
\bot \mapsto \emptyset, \ \ev \mapsto \{x \mid x \text{ mod } 2 = 0\}, \\
\od \mapsto \{x \mid x \text{ mod } 2 = 1\}, \ \top \mapsto \mathbb{Z}
\end{array} \right\} \\
\gamma_{\text{sign}} &= \left\{ \begin{array}{c}
\bot \mapsto \emptyset, \ 0 \mapsto \{0\}, \ \dot \mapsto \{x \mid x < 0\}, \\
\ddot \mapsto \{x \mid x > 0\}, \ \top \mapsto \mathbb{Z}
\end{array} \right\}
\end{align*}
\]

Consider the functions $\text{inc}, \text{dec}, \text{div} : \mathbb{Z} \to \mathbb{Z}$ defined respectively by $\lambda x.x + 1$, $\lambda x.x - 1$ and $\lambda x.x \div 2$. Possible approximations for $\text{Parity}$ and $\text{Sign}$ are given by:

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{op} & \bot & \od & \ev & \top \\
\hline
\text{inc}^A & \bot & \ev & \od & \top \\
\hline
\text{dec}^A & \bot & \ev & \od & \top \\
\hline
\text{div}^A & \bot & \top & \top & \top \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{op} & \bot & 0 & \dot & \ddot & \top \\
\hline
\text{inc}^A & \bot & + & \dot & \ddot & \top \\
\hline
\text{dec}^A & \bot & - & \dot & \ddot & \top \\
\hline
\text{div}^A & \bot & 0 & + & \ddot & \top \\
\hline
\end{array}
\]
Part I

Independence and Parallelism
Chapter 3

Independence in Constraint Logic Programming Languages

In this chapter we consider independence in the general context of the constraint logic programming (CLP) paradigm [86]. As mentioned in Chapter 1, this paradigm is a relatively recent proposal which has emerged as the natural combination of the constraint solving and logic programming paradigms. This combination makes it possible to parameterize a logic-based language and its semantics by a choice of domain of computation and constraints. In this context, traditional logic programming (LP) can be seen as an instance of CLP in which constraints are equations over terms and the domain of computation is the traditional Herbrand domain with the well known unification algorithm [132, 113].

As also mentioned in Chapter 1, independence refers to the conditions that the runtime behavior of the goals to be run in parallel must satisfy in order to guarantee the correctness and efficiency of the parallelization with respect to the sequential execution. Correctness is guaranteed if the answers obtained during the parallel execution are equivalent to those obtained during the sequential execution. Efficiency is guaranteed if the no “slow-down” property holds, i.e., if the parallel execution time is guaranteed to be shorter or equal than the sequential execution time.

Previous work in the context of traditional logic programming languages [36, 54, 73, 74, 75] has concentrated on defining sufficient conditions which ensure the preservation of the search space of the goals to be run in parallel. The reasons for this are twofold. Firstly, efficiency was ensured by requiring that the amount of work performed for computing the answers during the parallel execution be equal to that performed in the sequential execution. Such work was measured in terms of the number of non failure transitions performed. Secondly, it was shown that if a goal $g_1$ cannot change the
search space of a goal $g_2$ with respect to a given substitution $\theta$, then the correctness and efficiency (measured in the above terms) of their parallel execution with respect to the sequential execution of $(g_1 : g_2, \theta)$, is guaranteed.

Generalizing independence to arbitrary CLP languages and constraint solvers is not only interesting in itself, but also yields new insights into independence even for the Herbrand case. The CLP framework provides the tools for clarifying and formally defining the relationship between search space preservation and the notion of independence. In particular, we will show that search space preservation is, in the context of CLP languages, not only a sufficient but also a necessary condition for ensuring that both the intended solutions and the number of transitions performed do not change. Furthermore, we will also show that search space preservation is not enough to ensure independence in terms of total execution cost: while the number of transitions will certainly be constant if the search space is preserved, the cost of each reduction may not be preserved. The reason is that the cost of executing a sequence of primitive constraints may depend on the order in which those primitive constraints are considered. Thus, optimizations which vary the intended execution order established by the user, such as parallel execution, can actually cause execution to slow down. In order to ensure efficiency, we must therefore consider an additional issue – “independence of constraint solving” – which characterizes the properties of the constraint solver behavior when changing the order in which constraints are added. This issue has not risen previously because the standard unification algorithm, as usually implemented, is, in most practical cases, independent in this sense. However in the more general context of CLP, constraint solver independence need not hold.

As a final remark, note that the notions of independence given for LP were generally developed with one application, program parallelization, in mind. We will show that (both for LP and for CLP) different applications need, in order to ensure both the correctness and the efficiency of the transformations which they perform on the program, different levels of independence. Therefore, and for generality, we will formally define several concepts of independence for CLP languages at several levels, each of which will be shown to be “interesting” for a certain class of applications.

The generalization should be useful since the associated optimizations performed in the context of LP appear equally applicable to the context of constraints. Indeed, the cost of performing constraint satisfaction makes the potential performance improvements even larger. We look at three main applications. The first is and-parallelization of CLP programs. It is clear that adding constraints and running goals in parallel can dramatically improve performance. The second application is reordering of goals. This can transform a complex goal into a set of simple calculations or even simple tests.
This has been shown in [10] where primitive constraints and atoms are reordered. The concepts presented here extend this optimization to allow reordering of arbitrary goals. Our third application is intelligent backtracking. This can improve efficiency by avoiding re-execution of goals (and, therefore, constraint satisfaction operations) which have no relation with the failure being handled [2, 33].

The rest of the chapter proceeds as follows: in Section 3.1 the additional notation used throughout the chapter and some instrumental background on the CLP scheme is presented. Section 3.2 formally defines the relationship between search space preservation and independence. Section 3.3 presents several concepts of search space independence for CLP, each one useful for a class of applications. Several results on the implications of these definitions in terms of preservation of search space are then shown. Section 3.4 gives sufficient conditions that are easier to detect at run-time than the definitions of independence. Section 3.5 discusses the notion of independence for CLP at the solver level and discusses additional characteristics required of the solvers, offering some examples. Section 3.6 discusses several applications of the definitions and results presented in the previous sections. Finally, Section 3.7 shows our conclusions.

3.1 Background

In this section we present some background on the constraint logic programming scheme. In doing this, we will follow mainly [86, 87].

As before, upper case letters generally denote collections of objects, while lower case letters generally denote individual objects. $u, v, w, x, y, z$ will denote variables, $t$ will denote a term, $p, q$ will denote predicate symbols, $f$ will denote a function symbol, $a, h$ will denote atoms, $P, Q$ will denote programs and $g, G$ will denote goals. These symbols may be subscripted or have an over-tilde. $\tilde{x}$ denotes a sequence of distinct variables. $\exists_{\tilde{x}} \phi$ denotes the existential closure of the formula $\phi$ except for the variables $\tilde{x}$. $\exists_{\tilde{x}} \phi$ denotes the full existential closure of the formula $\phi$.

Syntactic Objects

Let $\Sigma$ and $\text{Var}$ denote a set of function symbols and a denumerable set of variables, respectively. Let $\Pi$ denote a set of predicate symbols such that $\Pi = \Pi_c \cup \Pi_P$ and $\Pi_c \cap \Pi_P = \emptyset$.

A primitive constraint has the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and $p \in \Pi_c$ is a predicate symbol. Every constraint is a conjunction of primitive constraints. The empty constraint is denoted $\epsilon$. An atom has the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and $p \in \Pi_P$. A literal is an atom or a primitive constraint, and will be usually
denoted by \( b \) or \( l \). A **CLP program** is a collection of clauses of the form \( h \leftarrow B \) where \( h \) is an atom (the head) and \( B \) is a sequence \( b_1, \ldots, b_n \) of literals (the body). We assume that the clauses are in normalized form, (or standard form), i.e., that all arguments in atoms are distinct variables, each variable occurring in at most one atom. A **goal** \( G \) (also denoted by \( g \)) is a sequence of literals.

A **renaming** is a mapping from \( \text{Var} \) to \( \text{Var} \). We let \( \text{Ren} \) be the set of renamings, and naturally extend renamings to mappings between atoms, clauses, and constraints. Syntactic objects \( s \) and \( s' \) are said to be **variants** if there is a \( \rho \in \text{Ren} \) such that \( \rho(s) \equiv s' \) (\( \equiv \) denotes syntactic equivalence for any syntactic object).

### Constraint domains

A **signature** defines two sets of function and predicate symbols (denoted by \( \Sigma \) and \( \Pi_e \) respectively) and associates an arity with each symbol. Given a signature \((\Sigma, \Pi_e)\), a \((\Sigma, \Pi_e)\)-**structure** \( D \) consists of:

- a non empty set \( D \) (domain of interpretation)
- an assignment for each function and relation in \((\Sigma, \Pi_e)\) of the elements on \( \{\{D\}\} \)
  which respects the arity of the symbols

A \((\Sigma, \Pi_e)\)-**theory** is a set of closed \((\Sigma, \Pi_e)\)-formulas. A \((\Sigma, \Pi_e)\)-structure \( D \) is a **model** of a \((\Sigma, \Pi_e)\)-theory \( T \) iff all formulas of \( T \) evaluate to true under the interpretation provided by \( D \). A \( D \)-model of a theory \( T \) is a model of \( T \) extending \( D \) (this requires that the signature of \( D \) be contained in the signature of \( T \)). \( T, D \models \phi \) denotes that the formula \( \phi \) is valid in all \( D \)-models of \( T \).

For any signature \((\Sigma, \Pi_e)\), let \( D \) be a \((\Sigma, \Pi_e)\)-structure and \( \mathcal{L} \) be a class of \((\Sigma, \Pi_e)\)-constraints. The pair \((D, \mathcal{L})\) is a **constraint domain**. We assume that:

- The terms and constraints in \( \mathcal{L} \) come from a first-order language.
- The binary predicate symbol \( = \) is contained in \( \Sigma \) and is interpreted as identity in \( D \).
- There are constraints in \( \mathcal{L} \) which are respectively identically true and identically false in \( D \).
- The class of constraints in \( \mathcal{L} \) is closed under variable renaming, conjunction and existential quantification.

The constraint domains are expected to support the following tests and operations on constraints:
1. Consistency or satisfiability of a constraint $c$: $\mathcal{D} \models \exists \mathcal{c}$

2. Implication or entailment of a constraint $c_1$ by a constraint $c_0$: $\mathcal{D} \models c_0 \rightarrow c_1$.

3. Projection of a constraint $c$ onto variables $\bar{x}$: $\mathcal{D} \models \exists_{\bar{x}} c$.

4. Detection that, given a constraint $c$, there is only one value that a variable $x$ can take that is consistent with $c$: $\mathcal{D} \models \exists x, \forall \bar{y} \ c(x, \bar{y}) \rightarrow x = z$. We say that $x$ is *definite* in $c$ and denote by $\text{def}(c)$ the set of definite variables in $c$.

Let us now present two properties that the constraint systems must satisfy.

**Definition 3.1.1** A constraint domain $(\mathcal{D}, \mathcal{L})$ is solution compact iff it satisfies the following conditions:

- for all $d \in D$ there exists $C \subseteq \mathcal{L}$ such that $\mathcal{D} \models \forall x \ x = d \leftrightarrow \land_{c \in C} c'(x)$
- for all $c \in \mathcal{L}$ there exists $C \subseteq \mathcal{L}$ such that $\mathcal{D} \models \forall \bar{x} \ c(\bar{x}) \leftrightarrow \lor_{c' \in C} c'(\bar{x})$.

**Definition 3.1.2** Let $(\mathcal{D}, \mathcal{L})$ be a constraint domain with signature $(\Sigma, \Pi_c)$, and $\mathcal{T}$ be a $(\Sigma, \Pi_c)$-theory. We say that $\mathcal{D}$ and $\mathcal{T}$ correspond on $\mathcal{L}$ iff:

- $\mathcal{D}$ is a model of $\mathcal{T}$, and
- for every constraint $c \in \mathcal{L}: \mathcal{D} \models \exists \mathcal{c}$ iff $\mathcal{T} \models \exists \mathcal{c}$

We say that $\mathcal{T}$ is satisfaction complete with respect to $\mathcal{L}$ iff for every constraint $c \in \mathcal{L}$ either $\mathcal{T} \models \exists \mathcal{c}$ or $\mathcal{T} \models \neg \exists \mathcal{c}$.

**CLP operational semantics**

The operational semantics [87] is presented as a transition system on states $(G, c, pc)$ where $G$ is a multi-set of literals, and $c$ and $pc$ are constraints ($(c \land pc)$ is called the *store*). Intuitively, $G$ is a collection of non executed literals, $c$ is formed by the constraints which are playing an *active role* and $pc$ is formed by the constraints playing a *passive role* (e.g. non linear equations in CLP($\mathcal{R}$)). There is one other state, denoted by fail. A *computation rule* which selects a transition type and an appropriate element of $G$ (if necessary) for each state is assumed as given. A *search rule* which selects (if necessary) a given clause of the program is also assumed as given.

The transition system is also parameterized by a predicate *consistent* and a function *infer*. The predicate *consistent*(c) expresses a test for consistency of $c$. Usually it is defined by: *consistent*(c) iff $\mathcal{D} \models \exists \mathcal{c}$, that is a complete consistency test. However, systems may employ a conservative but incomplete test: if $\mathcal{D} \models \exists \mathcal{c}$ then *consistent*(c)
holds but sometimes \( \text{consistent}(c) \) holds although \( \models \neg \exists c \). The function \( \text{infer}(c, pc) \) computes a new active constraint \( c' \) and passive constraint \( pc' \). It can be understood as relaxing \( pc \) in the presence of \( c \) to obtain more active constraints which are conjuncted to \( c \) to form \( c' \), \( pc \) being simplified to \( pc' \). It is required that \( D \models (c \land pc) \leftrightarrow (c' \land pc') \) so that the information is neither lost nor guessed by \( \text{infer} \). The transition rules in the transition system are:\(^1\)

- \( \langle G \cup a, c, pc \rangle \rightarrow_r \langle G \cup B, c, pc \land (a = h) \rangle \) if \( a \) is an atom selected by the computation rule, \( r \equiv h \leftarrow B \) is a clause of \( P \) renamed to new variables selected by the search rule, and \( h \) and \( a \) have the same predicate symbol.\(^2\)

- \( \langle G \cup a, c, pc \rangle \rightarrow_{r_1} \text{fail} \) if \( a \) is an atom selected by the computation rule and, for every clause \( r \equiv h \leftarrow B \) of \( P \), \( h \) and \( a \) have different predicate symbols.

- \( \langle G \cup c', c, pc \rangle \rightarrow_e \langle G, c, pc \land c' \rangle \) if \( c' \) is a constraint selected by the computation rule.

- \( \langle G, c, pc \rangle \rightarrow_i \langle G, c', pc' \rangle \) if \( (c', pc') = \text{infer}(c, pc) \).

- \( \langle G, c, pc \rangle \rightarrow_{t_1} \langle G, c, pc \rangle \) if \( \text{consistent}(c) \).

- \( \langle G, c, pc \rangle \rightarrow_{t_2} \text{fail} \) if \( \neg \text{consistent}(c) \).

The notions of derivation, derivation tree, and final state can be obtained as straightforward extensions of those defined for the LP context. The constraint \( c \land pc \) is said to be a partial answer to state \( s \) if there is a derivation from \( s \) to a state \( \langle G, c, pc \rangle \) and is said to be an answer if \( \langle G, c, pc \rangle \) is a final state and \( G \equiv \text{nil} \). Given a finite derivation with final state \( \langle \text{nil}, c, pc \rangle \), the derivation is successful if \( pc \) is empty, and it flounders otherwise. Thus, in this new context the answers are not always associated with successful derivations.

A CLP system is determined by the constraint domain and a detailed operational semantics. Let \( \rightarrow_{i_1} \equiv \rightarrow_{i} \rightarrow_{t} \) and \( \rightarrow_{i_2} \equiv \rightarrow_{i} \rightarrow_{t_1} \). A CLP system is quick-checking if transitions \( \rightarrow_r \) and \( \rightarrow_e \) are always followed by transitions \( \rightarrow_\| \) or \( \rightarrow_{i_2} \). A CLP system is progressive if, for every state with a nonempty collection of literals, every derivation of that state either fails, contains a \( \rightarrow_r \) transition or a \( \rightarrow_e \) transition. A CLP system is ideal if it is quick-checking, progressive, \( \text{infer} \) is defined by \( \text{infer}(c, pc) = (c \land pc, \epsilon) \) and the consistency test is complete.

\(^1\) In an abuse of notation we will use set union to express multi-set union.

\(^2\) The expression \( a = h \) is an abbreviation for the conjunction of the corresponding primitive equations.
Modifying the CLP operational semantics

We will particularize the operational semantics described above for CLP systems which satisfy the following requirements:

1. The computation rule implies left-to-right execution order. Thus, the element \(G\) of a state \(\langle G, c, pc \rangle\) will not be represented by a multi-set but by a sequence of literals.
2. The CLP systems are quick-checking and progressive.
3. There are no passive constraints i.e., \(infer(c, pc) = (c \land pc, \epsilon)\). Thus, states will be represented by \(\langle G, c \rangle\) where \(G\) is a sequence of literals, and \(c\) is the constraint store (always active).

The motivation behind assumptions (1) and (2) is to start from a particular sequential semantics with which to establish the appropriate comparisons. Note that most implemented CLP systems satisfy these assumptions. Assumption (3) is not really necessary. It has been required because the problems posed by passive constraints are subsumed by those appearing in dynamically scheduled languages, which are discussed in detail in Chapter 4.

Although we do not require the consistent function to be complete, it should satisfy the following two conditions. Firstly, it should not take variable names into account:

\[
\text{Let } \rho \in Ren. \text{ consistent}(\rho) \iff \text{consistent}(\rho(c))
\]

Secondly, if a constraint is said to be consistent, all constraints entailed are also consistent:

\[
\text{If } c \rightarrow c' \text{ and consistent}(c), \text{ then consistent}(c')
\]

The transition rules in the modified operational semantics are:

- \(\langle a : G, c \rangle \rightarrow_r \langle B :: G, c \land (a = h) \rangle\) if \(a\) is an atom, \(r \equiv h \leftarrow B\) is a clause of program \(P\) renamed to new variables selected by the search rule, and \(h\) and \(a\) have the same predicate symbol\(^3\).
- \(\langle a : G, c \rangle \rightarrow_{rf} \text{fail}\) if \(a\) is an atom and, for every clause \(r \equiv h \leftarrow B\) of \(P\), \(h\) and \(a\) have different predicate symbols.
- \(\langle c' : G, c \rangle \rightarrow_e \langle G, c \land c' \rangle\) if \(c'\) is a constraint and \(\text{consistent}(c \land c')\) holds.
- \(\langle c' : G, c \rangle \rightarrow_{ef} \text{fail}\) if \(c'\) is a constraint and \(\text{consistent}(c \land c')\) does not hold.

\(^3\)Note that the conjunction with the store is always consistent since we are considering normalized clauses.
Note that the conditions for applying each of the reduction rules are pairwise exclusive. This is necessary in order to simplify the definitions and theorems, and can always be achieved without loss of generality by grouping the application of some transition rules, as we will see in Section 4.3. Note also that answers will again correspond to successful derivations, since no floundering derivations can occur. We denote the set of answers to state $s$ for program $P$ by $\text{answer}_P(s)$, the partial answers by $\text{partial}_P(s)$, and the derivation tree by $\text{tree}_P(s)$.

### 3.2 Independence and Search Space Preservation

As mentioned in the introduction of this chapter, in traditional logic languages the concept of independence has always been defined in terms of search space preservation. However, we argue that the reasons behind this relationship have never been completely clarified due to the complications posed by the composition of substitutions when formalizing the concepts. In this section we will first briefly reconstruct, in the LP context, the reasoning followed by the authors of [73, 74, 75] when establishing the connections between independence and preservation of search space. We will then formalize the concept of search space preservation and its relationship with independence, in the more general context of CLP. Such formalization not only provides the proof of correctness for the intuition that preserving search space preserves both the correctness and efficiency of the and-parallel execution in LP, but also suggests better definitions of conditions for ensuring search space preservation.

#### 3.2.1 Independence in Logic Programs

The independent and-parallelism model [36, 54, 78, 75] aims at independently (i.e., guaranteeing no need for communication, and possibly in different environments) running in parallel as many goals as possible while maintaining correctness and efficiency with respect to the sequential execution. Correctness and efficiency were respectively defined as requiring that the answers obtained from the parallel execution be equivalent to those obtained in their sequential execution, and that the no “slow-down” property hold. Efficiency was approximated by requiring that the amount of work performed for computing the answers during the parallel execution be equal to that performed in the sequential execution. In this context, independence was defined as the characteristics that the goals had to satisfy in order to ensure the correctness and efficiency of their parallel execution.

The sequential framework assumed for LP is that presented in Section 2.2 equipped with a left-to-right computation rule. Formally, assume that at some point of the
execution $s \equiv \langle g_1 : \ldots : g_n, \theta \rangle$ is the current resolvent. The sequential SLD-resolution proof procedure with left-to-right selection rule would proceed as follows:

$$
\langle g_1 : \ldots : g_n, \theta \rangle \rightarrow^* \langle g_2 : \ldots : g_n, \theta_1 \rangle \rightarrow^* \langle g_3 : \ldots : g_n, \theta_2 \rangle \rightarrow^* \ldots
$$

Each $\rightarrow^*$ step represents the resolution of goal $g_i$ and its descendants in $s$ until they are all resolved and $g_{i+1}$ appears as the leftmost goal.

Assume that given the state $\langle g_1 : g_2 : G, \theta \rangle$ we want to execute $g_1$ and $g_2$ in parallel (the extension to a sequence of consecutive goals is straightforward). Then a possible execution scheme could be the following:

- execute $\langle g_1, \theta \rangle$ and $\langle g_2, \theta \rangle$ in parallel (in different environments) obtaining the answer substitutions $\theta_1$ and $\theta_2$ respectively,
- execute $\langle G, \theta_1, \theta_2 \rangle$.

It is assumed that the new variables introduced during the renaming steps in the parallel execution of the goals belong to disjoint sets. Also, note that the parallel framework can be applied recursively within the parallel execution of the goals in order to allow nested parallelism.

In this context, two main problems were detected. The first one, related to the variable binding conflict of [36], appears whenever during the parallel execution of $\langle g_1, \theta \rangle$ and $\langle g_2, \theta \rangle$ the same variable is bound to inconsistent values. Then, due to the definition of composition of substitutions [101, 4, 5] the answers obtained by the parallel execution can be different than those obtained by the sequential execution, thus affecting the correctness of the model, as shown in [75].

**Example 3.2.1** Consider the state $\langle p(x) : q(x), \epsilon \rangle$ and the following program:

$$
\begin{align*}
p(x) & \leftarrow x = a. \\
q(x) & \leftarrow x = b.
\end{align*}
$$

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

The second problem is due to the possibility of performing more work in the parallel execution than that performed during the sequential execution, thus affecting the efficiency of the model.

37
Example 3.2.2 Consider the state \( (p(x): q(x), \epsilon) \) and the following program:

\[
\begin{align*}
p(x) & \leftarrow x = a. \\
q(x) & \leftarrow x = b, \text{proc}, x = c.
\end{align*}
\]

where \text{proc} is very costly to execute.

While both the sequential and parallel execution will fail, their efficiency is quite different. While the sequential execution fails before executing \text{proc}, the parallel execution will first execute \text{proc} and then fail. □

A third problem was also detected whenever the goal to the left \((g_1 \text{ in the above model})\) has no answers, since then the amount of work performed by the parallel execution may be greater than that performed by the sequential execution and, thus, the no slow-down property may not hold. However, this problem was solved by assuming that the processor executing such a goal is able to kill the processors executing the goals to the right \((g_2 \text{ above})\), and that such processor has a higher priority than those executing goals to the right. As a result, the problem of ensuring the correctness and efficiency of the independent and-parallelism model was focused on ensuring that both the answers and the amount of work (measured in terms of number of non failure transitions in the derivation tree, in absence of the situation described above) obtained by the sequential and parallel execution of the goals, be the same.

The first solution proposed to ensure the two objectives was to only allow goals to be run in parallel if they do not share variables with respect to the current substitution [36]. This was formally defined in [73] as follows (and called “strict independence”):

Definition 3.2.1 [strict goal independence] Two goals \( g_1 \) and \( g_2 \) are said to be strictly independent with respect to a given substitution \( \theta \) iff \( \text{vars}(g_1\theta) \cap \text{vars}(g_2\theta) = \emptyset \). A collection of goals is said to be strictly independent for a given \( \theta \) iff they are pairwise strictly independent for \( \theta \). Also, a collection of goals is said to be strictly independent for a set of substitutions \( \Theta \) iff they are strictly independent for any \( \theta \in \Theta \). Finally, a collection of goals is said to be simply strictly independent if they are strictly independent for the set of all possible substitutions. □

The authors of [73] proved that if goals \( g_1 \) and \( g_2 \) are strictly independent with respect to a given substitution \( \theta \), then the parallel execution of \( \langle g_1, \theta \rangle \) and \( \langle g_2, \theta \rangle \) obtains the same answers as those obtained by the sequential execution of \( \langle g_1 : g_2, \theta \rangle \), and, in the absence of failure, parallel execution does not introduce any new work.

This sufficient condition is quite restrictive, since it can significantly limit the number of goals to be executed in parallel. However, as pointed out in [73], it has a very
important characteristic: strict independence is an a priori condition (i.e., it can be tested at run-time before executing the goals).

Due to the restrictive nature of the notion of strict independence, there have been several attempts to identify a more general sufficient condition. The intuition behind such generalizations is that goals sharing variables could still be run in parallel when the bindings established for those shared variables satisfy certain characteristics. This was informally discussed in [54, 155, 157], formally defined in [73], refined in [74], and further refined in [75] as follows:

**Definition 3.2.2** [v- and nv-binding] A binding $x/t$ is called a v-binding if $t$ is a variable, otherwise it is called an nv-binding.

**Definition 3.2.3** [non-strict independence] Consider a collection of goals $g_1, \ldots, g_n$ and a substitution $\theta$. Consider also the set of shared variables $SH = \{v \mid \exists i, j, 1 \leq i, j \leq n, i \neq j, v \in (var(g_i \theta) \cap var(g_j \theta))\}$ and the set of goals containing each shared variable $G(v) = \{g_i \theta \mid v \in var(g_i \theta), v \in SH\}$. Let $\theta_i$ be any answer substitution to $g_i \theta$. The given collection of goals is non-strictly independent for $\theta$ if the following conditions are satisfied:

- $\forall v \in SH$, at most the rightmost $g \in G(v)$, say $g_j \theta$, nv-binds $v$ in any $\theta_j$;
- for any $g_i \theta$ (except the rightmost) containing more than one variable of $SH$, say $v_1, \ldots, v_k$, then $v_i \theta_i, \ldots, v_k \theta_i$ are strictly independent.

Intuitively, the first condition of the above definition requires that at most one goal further instantiate a shared variable. The second condition eliminates the possibility of creating aliases (of different shared variables) during the execution of one of the parallel goals which might affect goals to the right.

At this point it was noticed that, due to the definition of the composition of substitutions, incorrect answers could be obtained even when there was no variable binding conflict for the shared variables.

**Example 3.2.3** Consider the state $\langle p(x, y) : q(y), \epsilon \rangle$ and the program:

```
p(x, y) ← x = y.
q(x) ← x = a.
```

It is easy to check that $p(x, y)$ and $q(y)$ are non-strictly independent for $\epsilon$. However, if we run $\langle p(x, y), \epsilon \rangle$ we obtain $\theta_p = \{x/z, y/z\}$. If we now execute $\langle q(y), \theta_p \rangle$ we obtain the substitution $\theta = \{x/a, y/a, z/a\}$. If, instead we execute $\langle q(y), \epsilon \rangle$ we obtain
\( \theta_q = \{ y/a \} \) thus ending with their composition \( \theta_p \theta_q = \{ x/z, y/z \} \) as final substitution. This answer is obviously different from the \( \theta \) obtained by the sequential execution, thus yielding an incorrect result. \( \Box \)

As noticed in both [74] and [75], this could be easily solved by defining a “parallel composition” which avoids these problems. Such parallel composition was defined in terms of “solving” the equations associated with the substitutions being composed. However, adopting a new definition of composition would have required a revision of well known results in logic programming, which rely on the standard definition. As a result, the authors adopted a different solution which involved a renaming transformation. Informally, the renaming transformation of two goals \( g_1 \) and \( g_2 \) for a substitution \( \theta \), involves applying the substitution to both goals, eliminating any shared variables in the resulting goals by renaming all their occurrences (so that no two occurrences in different goals have the same name), and adding some unification goals to reestablish the lost links (for a formal definition see [75]).

**Example 3.2.4** Consider the collection of goals \((r(x, z, x), s(x, w, z), p(x, y), q(y))\) in a state (we consider \( \theta \) already applied to the goals). According to the definition of renaming transformation, we will write this new collection of goals as follows:

\[
r(x, z, x), s(x', w, z'), p(x'', y), q(y'), x = x', x = x'', y = y', z = z'.\]

Note that the first goal always remains unchanged. Goals of the form \( x = x' \) above were called “back-binding” goals (denoted by \( BB \)) and are related to the back-unification goals defined in [94], and the closed environment concept of [37]. In this context, the parallel framework described above was redefined as follows:

Assume that given the state \( \langle g_1 : g_2 : G, \theta \rangle \) we want to execute \( g_1 \) and \( g_2 \) in parallel (the extension to more than two goals is straightforward). Then, the execution scheme was defined as follows:

- apply the renaming transformation to \( g_1 \theta, g_2 \theta \) obtaining \( g'_1, g'_2, BB \),
- execute \( \langle g'_1, \epsilon \rangle \) and \( \langle g'_2, \epsilon \rangle \) in parallel (in different environments) obtaining the answer substitutions \( \theta_1 \) and \( \theta_2 \) respectively,
- execute \( \langle BB, \theta_1, \theta_2 \rangle \) obtaining the answer substitution \( \theta_3 \),
- execute \( \langle G, \theta_1 \theta_3 \rangle \).

As before, it is assumed that the new variables introduced during the renaming steps in the parallel execution belong to disjoint sets.
Once the parallel framework was redefined, the notions of correctness and efficiency were also reconsidered. Correctness was not a significant problem since, in general, the answers provided by the parallel executions were the same (up to renaming) as the answers obtained in the sequential execution. Only an infinite derivation in the execution of \( \langle d'_2, \epsilon \rangle \) would yield a change, if there is no such infinite derivation in the sequential execution due to the effect of some answer to \( \langle g_1, \theta \rangle \). However, since this was a particular case in which efficiency was also affected, the correctness problem was ignored in the knowledge that if efficiency was achieved this case could not happen and therefore correctness would also be ensured.

Inefficiency was then assumed to come from two sources. Firstly, due to a larger branch in the derivation tree associated with the parallel execution of \( \langle d'_2, \epsilon \rangle \), since such a tree would obviously imply more work. This was the point in which the notion of search space preservation was introduced. Unfortunately, such notion was never formally defined, the intuitive idea given for the preservation of the search space being the following: the search space of two states are the same if their associated derivation trees have the same “shape” [75]. This concept was later (in some sense erroneously) identified with the preservation of the number of non failure nodes in the respective derivation trees. Secondly, due to answers obtained during the parallel execution which when executing the back-bindings yield a failure, since this would again increase the work (backtracking, finding another answer, etc). Initially, concentrating on the success of the back-bindings introduced some confusion since it was easy to believe that if such bindings always succeed then the efficiency (and thus the correctness) of the parallel model was ensured. However, as pointed out in [75], this does not ensure the preservation of the amount of work in failed derivations.

Although the work developed in [75] does provide the basic results for LP, it is our thesis that the problems associated to the composition of substitutions, which yield the introduction of the renaming transformation, were one of the main reasons which prevented the clarification of all the issues introduced above. One the one hand the authors did not focus on the original variable binding conflict which, as we will show later, is the main source of the problems and provides the correct intuition for generalizing independence to the CLP paradigm. On the other hand they were just one step behind the definition of not only sufficient but also necessary conditions which ensure search space preservation and its relation with the success of back-bindings.

In the following section we will present such formalization in the context of CLP languages.
3.2.2 Independence in Constraint Logic Programs

We start from the observation that the view of composition of substitutions in CLP corresponds exactly with the "parallel composition" needed in [74]. What in the LP scheme would imply a reconsideration of the standard theory and results comes for granted in the CLP scheme. Therefore, we avoid the renaming transformation and redefine the parallel model as follows. Assume that given the state $(g_1 : g_2 : G, c)$ we want to execute $g_1$ and $g_2$ in parallel (the extension to more than two goals is straightforward). Then the execution scheme is the following:

- execute $(g_1, c)$ and $(g_2, c)$ in parallel (in different environments) obtaining the answer constraints $c_1$ and $c_r$, respectively,
- obtain $c_s$ as the conjunction of $c_1 \land c_r$,
- execute $(G, c_s)$.

As before, and in order to avoid problems when conjoining $c_1$ and $c_r$, it is assumed that the new variables introduced during the renaming steps in the parallel execution of the goals belong to disjoint sets.

In this context, the correctness of the parallel model requires that for each answer $c_s$ obtained in the sequential execution there exist two answers $c_1$ to $(g_1, c)$ and $c_r$ to $(g_2, c)$ which when conjuncted provide the same answer. Formally:

**Definition 3.2.4** Let $(g_1 : g_2 : G, c)$ be a state and $P$ be a program. The parallel execution of $g_1$ and $g_2$ is correct iff for every $c_1 \in answers_P((g_1, c))$ there exists a renaming $\rho \in Ren$ and a bijection which assigns to each $c_s \in answers_P((g_2, c_1))$ an answer $c_r \in answers_P((g_2, c_1))$ with $c_s \leftrightarrow c_1 \land \rho(c_r)$. ■

The efficiency of the parallel model requires that, in absence of failure (i.e., when the goals to the left have at least one answer), the amount of work performed during the parallel execution be less or equal to that performed during the sequential execution. We will assume that the application of a particular transition rule has the same cost, independently of the state to which the transition is applied. Therefore, in order to obtain the amount of work performed in the execution of a given state $s$, we only need

---

4The suffix $s$ will be associated to the arguments of the states obtained during the sequential execution.

5The suffix $r$ will be associated to the arguments of the states obtained during the parallel execution of the goal to the right.

6This has been traditionally assumed in the context of LP. As we will discuss in Section 3.5 this cannot be assumed in the more general context of CLP.
to know the cost of applying each particular transition rule and number of times in which each transition rule has been applied. Let $TR$ be the set of different transition rules that can be applied. Let $s$ be a state and $N(i, s)$ be the number of times in which a particular transition rule $i \in TR$ has been applied in $\text{tree}_P(s)$. Let $K(i)$ be the cost of applying a particular transition rule $i \in TR$, and assume that such cost is always greater than zero.

**Definition 3.2.5** Let $\langle g_1 : g_2 : G, c \rangle$ be a state and $P$ be a program. The parallel execution of $g_1$ and $g_2$ is efficient iff for every $c_1 \in \text{answers}_P(\langle g_1, c \rangle)$:

$$\sum_{i \in TR} K(i) * N(i, \langle g_2, c \rangle) \leq \sum_{i \in TR} K(i) * N(i, \langle g_2, c_1 \rangle)$$

Note that the amount of work performed in conjoining the answers obtained from the parallel execution is not taken into account. We consider the cost of such operation as one of the overheads associated with the parallel execution (as creation of processors, scheduling, etc). Let us now focus on the definition of search space preservation and its relationship with the preservation of correctness and efficiency of the parallel execution with respect to the sequential execution. We assume that nodes in the derivation tree are labeled with their path. We say that two nodes $n$ and $n'$ in the derivation trees of states $s$ and $s'$, respectively, and with the same path correspond if either they are the roots of the tree (i.e., $n \equiv s$ and $n' \equiv s'$) or they have been obtained by applying the same reduction rule.

**Definition 3.2.6** States $s$ and $s'$ have the same search space for program $P$ iff there exists a (total) bijection which assigns to each node in $\text{tree}_P(s)$ its corresponding node in $\text{tree}_P(s')$. ■

The properties of the search space preservation and the particular characteristics of the two states, $\langle g_2, c \rangle$ and $\langle g_2, c_1 \rangle$, for which the search space preservation is required (the initial sequence of literals is the same and $c_1 \rightarrow c$) allow us to ensure the following result:

**Theorem 3.2.7** Let $\langle g_1 : g_2 : G, c \rangle$ be a state and $P$ a program. The parallel execution of $g_1$ and $g_2$ is correct if for every $c_1 \in \text{answers}_P(\langle g_1, c \rangle)$: the search spaces of $\langle g_2, c \rangle$ and $\langle g_2, c_1 \rangle$ are the same for $P$. ■

The proof comes directly from the following two lemmas.

**Lemma 3.2.8** Let $\langle g_2, c_1 \rangle$ and $\langle g_2, c \rangle$ be two states with an identical sequence of literals, and $P$ a program. There exists a renaming $\rho \in \text{Ren}$ such that for every two non
failure nodes $s \equiv \langle G_s, c_s \rangle$ and $r \equiv \langle G_r, c_r \rangle$ with the same path in $\text{tree}P(\langle g_2, c_1 \rangle)$ and $\rho(\text{tree}P(\langle g_2, c \rangle))$, respectively: $G_s \equiv G_r$.

**Proof:** Consider the trees obtained by obtaining all direct children of $\langle g_2, c_1 \rangle$ and $\langle g_2, c \rangle$ (i.e., the first level of the derivation tree). If the leftmost literal in $g_2$ is a constraint and $\to_e$ is applied, the renaming is the identity. Otherwise, the leftmost literal in $g_2$ is an atom and one or more $\to_r$ are applied. Let $R$ be the set of rules in $P$ which can be selected for such transition. Let $\rho_1$ and $\rho_2$ be the renamings applied to them before performing the transitions for $\langle g_2, c_1 \rangle$ and $\langle g_2, c \rangle$, respectively. By assumption, while the domain of $\rho_1$ and $\rho_2$ is the same set, their ranges are disjoint sets. Then $\rho$ can be obtained as $\rho_1 \rho_2^{-1}$. The theorem is thus obtained by applying this result recursively to the nodes in the derivation tree.

Note that, as constructed, the domain of $\rho$ is the range of $\rho_2$ and therefore $\rho(\text{tree}P(\langle g_1, c_1 \rangle)) \equiv \text{tree}P(\langle g_1, c_1 \rangle)$ and $\rho(\langle g_2, c \rangle) \equiv \langle g_2, c \rangle$. In the rest of this chapter we assume that $\rho$ satisfy such characteristic.

**Lemma 3.2.9** Let $\langle g_2, c_1 \rangle$ and $\langle g_2, c \rangle$ be two states with an identical sequence of literals such that $c_1 \to c$. Let $P$ be a program and $\rho$ be a renaming satisfying Lemma 3.2.8. Then, for every two nodes $s \equiv \langle G_s, c_s \rangle$ and $r \equiv \langle G_r, c_r \rangle$ with the same path in $\text{tree}P(\langle g_2, c_1 \rangle)$ and $\rho(\text{tree}P(\langle g_2, c \rangle))$, respectively: $c_s \leftrightarrow c_1 \land c_r$.

**Proof:** Let $\rho$ be a renaming satisfying the conditions in Lemma 3.2.8. Since by Lemma 3.2.8 the sequences of literals of all parents of $s$ are identical to those of all parents of $r$ with the same path, the constraints added to $c_1$ and to $c$, yielding $c_s$ and $c_r$ respectively, have been the same. Since by assumption $c_1 \to c$, and therefore $c_1 \leftrightarrow c_1 \land c_r$, it is clear that $c_s \leftrightarrow c_1 \land c_r$.

Note that even if the consistent function associated with the underlying constraint system is not complete, and thus both $c_s$ and $c_1 \land c_r$ possibly entail fail, the lemma is satisfied.

Those two lemmas, and the fact that search space preservation implies a bijection among answers, allow us to prove that search space preservation is sufficient for ensuring the correctness of the parallel execution. However, it is important to note that search space preservation is not necessary for ensuring correctness since it is possible that although nodes in successful derivations correspond, there exists at least two nodes in failure derivations which do not correspond.

Ensuring that efficiency is also guaranteed is even easier due to the definition of search space:

44
Theorem 3.2.10 Let $\langle g_1 : g_2 : G, c \rangle$ be a state, $P$ be a program, and $c_1 \in \text{answers}_P(\langle g_1, c \rangle)$. If the search spaces of $\langle g_2, c \rangle$ and $\langle g_2, c_1 \rangle$ are the same for $P$, then $\sum_{i \in \text{TR}} K(i) \ast N(i, \langle g_2, c \rangle) = \sum_{i \in \text{TR}} K(i) \ast N(i, \langle g_2, c_1 \rangle)$.

Proof: By definition of search space preservation, there exists a bijection among the nodes in $\text{tree}_P(\langle g_2, c \rangle)$ and $\text{tree}_P(\langle g_2, c_1 \rangle)$ which are obtained by applying the same transition rule. Thus, for any $i \in \text{TR}: N(i, \langle g_2, c \rangle) = N(i, \langle g_2, c_1 \rangle)$.

As before, note that search space preservation is not necessary for ensuring efficiency since, for example, if different transition rules have the same cost, even if the nodes in the trees do not correspond, the total cost can be the same. Then, we can state the following:

Theorem 3.2.11 Let $\langle g_1 : g_2 : G, c \rangle$ be a state and $P$ a program. The parallel execution of $g_1$ and $g_2$ is efficient if for every $c_1 \in \text{answers}_P(\langle g_1, c \rangle)$: the search spaces of $\langle g_2, c \rangle$ and $\langle g_2, c_1 \rangle$ are the same for $P$.

Thus we have already proved that search space preservation is sufficient for ensuring the correctness and also for ensuring the efficiency of the parallel execution. However, we can go further and show that it is in fact necessary for ensuring that both correctness and efficiency hold. The following lemmas are instrumental for this result.

Lemma 3.2.12 Let $\langle g_2, c_1 \rangle$ and $\langle g_2, c \rangle$ be two states with an identical sequence of literals such that $c_1 \rightarrow c$. Let $P$ be a program. Then, for every two nodes $s$ and $r$ with the same path in $\text{tree}_P(\langle g_2, c_1 \rangle)$ and $\text{tree}_P(\langle g_2, c \rangle)$, respectively: $s$ and $r$ have been obtained with the same transition rule if and only if either $s \equiv r \equiv \text{fail}$ or they are both non-failure nodes.

Proof: On the one hand, if $s$ and $r$ have been obtained by the same transition rule, by definition of the operational semantics, either both are identical to $\text{fail}$ or both are non-failure. On the other hand, let $s'$ and $r'$ be the parents of $s$ and $r$, respectively. By Lemma 3.2.8, if the leftmost literal in the sequence of literals in $s'$ is an atom (constraint) then the leftmost literal in the sequence of literals in $r'$ must also be an atom (constraint). Then, by definition of the operational semantics, if either $s \equiv r \equiv \text{fail}$ or they are both non-failure nodes, they must have been obtained by applying the same transition rule.

Lemma 3.2.13 Let $\langle g_2, c_1 \rangle$ and $\langle g_2, c \rangle$ be two states with an identical sequence of literals such that $c_1 \rightarrow c$ and the search spaces of $\langle g_2, c_1 \rangle$ and $\langle g_2, c \rangle$ are different for program $P$. Then, there exists a bijection which assigns to each node $s$ in $\text{tree}_P(\langle g_2, c_1 \rangle)$ for which there is no corresponding node in $\text{tree}_P(\langle g_2, c \rangle)$, a node $r$ in $\text{tree}_P(\langle g_2, c \rangle)$ with the same path, such that $s$ and $r$ have been obtained applying the $\rightarrow c$ and $\rightarrow c$ transition rule, respectively, and the parents of $s$ and $r$ correspond.
Proof: Let us assume that the search space is not preserved and that $s$ is the first node in its branch for which there is no corresponding node. Let $s'$ be the parent of $s$. By assumption, $s'$ has a corresponding node $r'$ in $\text{tree}_P((g_2,c))$. Since $s'$ is non-failure, by Lemma 3.2.12, $r'$ must be also non-failure. Thus, there must exist a node $r$ in $\text{tree}_P((g_2,c))$ with the same path as $s$. The same reasoning applies for $r$, since by assumption its parent has a corresponding node. Thus, if there exists such $s$, $r$ exists and vice versa. By definition of search space preservation and Lemma 3.2.12, if $s$ and $r$ do not correspond, one must be fail and the other must not. Let $s' \equiv \langle G'_s, c'_s \rangle$, $r' \equiv \langle G'_r, c'_r \rangle$, and $\rho$ be the renaming satisfying the conditions in Lemma 3.2.8. By Lemma 3.2.8, $G'_s \equiv \rho(G'_r)$. If the first literal in $G'_s$ is an atom, both $s$ and $r$ must have been obtained by applying either $\rightarrow_r$ or $\rightarrow_r$, and thus correspond. Therefore, the first literal in $G'_s$ must be a constraint $c'$. If $s$ have been obtained by applying the $\rightarrow_c$ transition rule, then $\text{consistent}(c'_s \land c')$ must hold. However, by Lemma 3.2.9 $c'_s \leftrightarrow c'_1 \land \rho(c'_r)$. Therefore, $\text{consistent}(c' \land \rho(c'_r))$ must also hold, and thus $r$ must also be obtained by applying the $\rightarrow_c$ rule. Thus, the only possibility is that while $\text{consistent}(c'_s \land c')$ does not hold, $\text{consistent}(c' \land \rho(c'_r))$ holds. Thus, if have been obtained by applying the $\rightarrow_c$ and $\rightarrow_r$ transition rules, respectively. Finally, $s$ must be the first node for which there is no corresponding node, since no node has fail as parent.

As a result, we can conclude that the only way in which the search spaces of $(g_2,c_1)$ and $(g_2,c)$, with $c_1 \rightarrow_c c$, can be different for a program $P$, is by pruning some branch of $\text{tree}_P((g_2,c))$. Now we can state the following:

**Theorem 3.2.14** Let $(g_1 : g_2 : G,c)$ be a state and $P$ a program. The parallel execution of $g_1$ and $g_2$ is correct and efficient iff for every $c_1 \in \text{answers}_P((g_1,c))$ the search spaces of $(g_2,c)$ and $(g_2,c_1)$ are the same for $P$.

**Proof:** Since we have already proved that search space preservation is sufficient, let us focus on the necessary condition. Let us reason by contradiction and assume that the parallel execution is correct and efficient but there exists at least one $c_1 \in \text{answers}_P((g_1,c))$ for which the search spaces of $(g_2,c)$ and $(g_2,c_1)$ are not the same for $P$. By Lemma 3.2.13 we can ensure that for every node in $\text{tree}_P((g_2,c_1))$ obtained with one of the transition rules in $\{\rightarrow_r, \rightarrow_r, \rightarrow_r\}$, there exists a corresponding node in $\text{tree}_P((g_2,c))$ which has been obtained with the same transition rule. Thus, for every $i \in \{\rightarrow_r, \rightarrow_r, \rightarrow_r\}$: $N(i,(g_2,c_1)) \leq N(i,(g_2,c))$. Also, for every node $s$ in $\text{tree}_P((g_2,c_1))$ obtained with the $\rightarrow_r$ transition rule, either it has a corresponding node in $\text{tree}_P((g_2,c))$ or, by Lemma 3.2.13 there exists a node $r$ in $\text{tree}_P((g_2,c_1))$ with the same path, which have been obtained applying the $\rightarrow_c$ transition rule. By definition
of correctness there exists a bijection among answer nodes, i.e. nodes in successful derivations. Thus $r$ must be non failure, and the branches starting at $r$ must be either infinite or failure. Thus the amount of work performed for obtaining $r$ and its children is greater than that performed for obtaining $s$. But then the parallel execution is not efficient, which is in contradiction with the assumption. ■

Note that this implies that, in absence of failure, the amount of work performed during the parallel execution is in fact equal (never less) than that performed in the sequential execution, the speedup coming from the parallel execution of such work. From the results above we can also clarify two of the points mentioned in the summary of the results for LP. Let $\#nodes_P(s)$ be the number of non failure nodes in the derivation tree of state $s$ for program $P$.

**Corollary 3.2.15** Let $\langle g_2, c_1 \rangle$ and $\langle g_2, c \rangle$ be two states with an identical sequence of literals such that $c_1 \rightarrow c$. Let $P$ be a program. The search spaces of $\langle g_2, c \rangle$ and $\langle g_2, c_1 \rangle$ are the same for $P$ iff $\#nodes_P(\langle g_2, c \rangle) = \#nodes_P(\langle g_2, c_1 \rangle)$. ■

The proof comes directly from Lemma 3.2.12. This can explain why preservation of the search space and of the number of non failure nodes were identified. However, as we will see later, this identification cannot be done when dynamically scheduled languages are taken into account, since then a more constrained store $c_1$ can both prune and enlarge the search space.

The second confusing point, clarified in [75] for LP, was related to the success of the back bindings in the parallel framework based on the renaming transformation, can also be derived for CLP:

**Corollary 3.2.16** Let $\langle g_1 : g_2 : G, c \rangle$ be a state, and $P$ a program. If for every $c_1 \in answers_P(\langle g_1, c \rangle)$, the search spaces of $\langle g_2, c \rangle$ and $\langle g_2, c_1 \rangle$ are the same for $P$, the back bindings resulting from the parallel execution of $g_1$ and $g_2$ in the parallel model requiring the renaming transformation will always succeed. ■

The proof comes directly from Theorem 3.2.7. Furthermore, inherited from Theorem 3.2.7, the success of the back bindings does not guarantee the preservation of the search space, thus confirming the results in [75].

At this point it is clear that, following the definitions of correctness and efficiency given above, search space preservation ensures both the correctness and efficiency of the parallel execution of independent goals. Furthermore, that search space preservation is not only a sufficient but also a necessary condition for ensuring both efficiency and correctness. However, there are still two issues related to the assumptions made when
ensuring efficiency. Firstly, we have assumed that \( g_1 \) has at least one answer. If this is not true, the amount of work during the parallel execution might be increased. Such increment will depend on how the implemented system handles such situations. However, given the results above, if we assume the behavior of the system in case of failure proposed in [75], the same results can be obtained, thus ensuring efficiency also for those cases. Secondly, we have also assumed that the amount of work involved in applying a particular transition rule is independent of the state to which the rule is applied. Thus, there is one point which has not been taken into account, namely the changes in the amount of work involved when applying a particular transition rule to states with different constraint stores. As we will see in Section 3.5, in the context of CLP languages this change can be significant, actually possibly yielding a slow-down.

### 3.3 Levels of Independence

In this section we will formally define the concept of independence for CLP languages at several levels. As mentioned in the previous section, independence was also defined at several levels in LP, starting from the most restrictive definitions which ensure search space preservation, and then extending such conditions in order to enlarge the number of goals which can be considered independent, while still preserving the search space. We will do the opposite: start from a quite general concept of independence which is not sufficient for ensuring search space preservation, and then progressively restrict such definition in order to ensure search space preservation. The reason for this is to show that such definitions are interesting by themselves rather than just a condition for the correctness and efficiency of the parallel execution of goals, due to their importance for a certain class of applications.

#### 3.3.1 Weak Independence

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

**Example 3.3.1** Consider the following fragment of a CLP(\( \mathbb{R} \)) program:

\[
\begin{align*}
p(x,y) & : - x > 0. \\
p(x,y) & : - x = 3, x = y. \\
p(x,y) & : - x > 7. \\
q(x,y) & : - x < 5, y = 2. \\
q(x,y) & : - x > 1.
\end{align*}
\]
Figure 3.1:

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

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

**Definition 3.3.1** [weak independence] Goals \( g_1 \) and \( g_2 \) are weakly independent for constraint \( c \) and program \( P \) iff

\[
\forall c_1 \in \text{answer}_P(\langle g_1, c \rangle) \text{ and } \forall c_r \in \text{answer}_P(\langle g_2, c \rangle) : \text{consistent}(c_1 \land c_r)
\]

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

Note that, according to this definition, goals which fail (those for which the set of answers is empty) for a given constraint are weakly independent of all other goals. Also, note that this concept can be associated with that of preservation of the search space in the successful derivations, and thus with the success of the back-bindings.

**Lemma 3.3.2** Goals \( g_1 \) and \( g_2 \) are weakly independent for constraint \( c \) and program \( P \) iff \( \forall c_1 \in \text{answer}_P(\langle g_1, c \rangle) : \) there exists a bijection which assigns to each node in a successful branch of \( \text{tree}(\langle g_2, c \rangle) \) a corresponding node in a successful branch of \( \text{tree}(\langle g_2, c_1 \rangle) \).

**Proof:** First, assume that the bijection exists. Since the nodes are non failure, by Lemma 3.2.9 \( g_1 \) and \( g_2 \) must be weakly independent for \( c \) and \( P \). Second, assume
that \( g_1 \) and \( g_2 \) are weakly independent for \( c \) and \( P \). By Lemma 3.2.13 for all non failure nodes, and in particular those in successful branches of \( \text{treee}(g_2, c_1) \), there exists a corresponding node in a successful branch of \( \text{treee}(g_2, c) \). Also, by assumption of weak independence, for each node \( r \equiv (G_r, c_r) \) in a successful branch of \( \text{treee}(g_2, c) \) \( \text{consistent}(c_1 \land c_r) \) holds. Let \( \rho \) be a renaming satisfying the conditions in Lemma 3.2.8. By construction, \( \rho(c_1) \equiv c_1 \) and thus \( \text{consistent}(c_1 \land \rho(c_r)) \) must also hold. By Lemma 3.2.9, the consistency tests for obtaining the nodes with the same path as \( r \) are performed over a constraint \( c_s \) satisfying \( c_s \leftrightarrow c_1 \land \rho(c_r) \) and thus, \( \text{consistent}(c_s) \) must also hold. Therefore, by Lemma 3.2.12 for each such \( r \) node there is a corresponding node, in a successful branch.

Detecting this kind of independence can, in principle, be useful for performing optimizations which are based on determination of producer-consumer relationships, such as intelligent backtracking. This is based on the following result:

**Theorem 3.3.3** Let \( g_1 : \cdots : g_n \) be a collection of weakly independent goals for constraint \( c \) and program \( P \). Let \( g_i, 1 \leq i \leq n \) be a goal such that there exists \( c_1 \in \text{answers}_P(g_1 : \cdots : g_{i-1}, c) \) with \( \text{answers}_P(g_i, c_1) = \emptyset \). Then, for every \( c_2 \in \text{answers}_P(g_1 : \cdots : g_{i-1}, c) : \text{answers}_P(g_i, c_2) = \emptyset \).

The proof comes directly from the Lemma 3.3.2. Thus, once there are no answers for goal \( g_i \), execution can safely backtrack to the choice-point placed just before \( g_i \), skipping all the choice-points in between. This application will be further discussed in Section 3.6.4.

However, from the results of the previous section, it is clear that weak independence is not sufficient for ensuring search space preservation (as the success of the back-bindings cannot either), since only successful derivations of the goals have been considered and the search space can also be affected through interactions with derivations failed or infinite derivations.

**Example 3.3.2** Consider the previous example. Assume that we start from the state \( \langle p(x, y) : q(x, y), \{y = 1\} \rangle \). It is clear that the search space associated to \( \langle q(x, y), y = 1 \land x > 7 \rangle \) is less than that associated to \( \langle q(x, y), \{y = 1\} \rangle \) since, the derivation in which \( x < 5 \) appears would fail earlier – as soon as \( x < 5 \) is checked for consistency with the store.

### 3.3.2 Strong Independence

We now formally define a more restrictive concept of independence, in the spirit suggested above of taking into account all partial answers, which we will call strong inde-
Definition 3.3.4 [strong independence] Goal $g_2$ is strongly independent of goal $g_1$ for constraint $c$ and program $P$ iff

$$\forall c_1 \in \text{answer}_P((g_1, c)) \text{ and } \forall c_r \in \text{partial}_P((g_2, c)) : \text{consistent}(c_1 \land c_r)$$

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

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

Example 3.3.3 In the example given in Figure 1, $p(x, y)$ is strongly-independent of $q(x, y)$ for $c \equiv \{y = 1\}$ since all answers to $(q(x, y), c)$ are consistent with partial answers of $(p(x, y), c)$. However, $q(x, y)$ is not strongly-independent of $p(x, y)$ for the same constraint $c$. 

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

We will now show some properties which hold for strongly independent goals. The main result is that goal $g_2$ is independent of $g_1$ for a given constraint $c$ if and only if the search space is preserved.

Theorem 3.3.5 Goal $g_2$ is strongly independent of goal $g_1$ for constraint $c$ and program $P$ iff

$$\forall c_1 \in \text{answer}_P((g_1, c)) : \text{the search spaces of } (g_2, c) \text{ and } (g_2, c_1) \text{ are the same.}$$

Proof: If search space is preserved, by Lemma 3.2.9 $g_1$ and $g_2$ must be strongly independent for $c$ and $P$. On the other hand, assume that $g_1$ and $g_2$ are strongly independent for $c$ and $P$. By Lemma 3.2.13 if the search spaces are not preserved, there must exist a node $s \equiv \text{fail}$ in tree$P((g_2, c_1))$ obtained by applying $\rightarrow_{cf}$ and a node $r \equiv (G_r, c_r)$ obtained by applying $\rightarrow_c$ in tree$P((g_2, c))$ with the same path, such that their parents, correspond. Let $\rho$ be a renaming satisfying the conditions in Lemma 3.2.8. By construction, $\rho(c_1) \equiv c_1$ and thus, by strong independence, consistent$(c_1 \land \rho(c_r))$ must also hold. By Lemma 3.2.8 and Lemma 3.2.9, the consistent test performed for obtaining $s$ was applied to a constraint $c_s \leftrightarrow c_1 \land \rho(c_r)$ and thus, consistent$(c_s)$ must also hold. This is in contradiction with $s$ being fail.
This theorem ensures that strong independence is not only sufficient but also necessary for ensuring preservation of search space. It is important to note that if we define strong independence in terms of the full existential closure and the consistent function associated with the underlying constraint system is not complete, then strong independence would be sufficient but not necessary. In any case, the correctness and efficiency of the parallel execution of a set of strongly independent goals with respect to the the original left to right execution is ensured.

When reordering goals it is difficult to give simple yet general conditions which ensure that the reordering of two goals reduces the search space. However, one simple condition that ensures that the reordering does not increase the search space is that the rightmost goal is “single solution” and strongly independent of the leftmost goal.\(^7\)

**Definition 3.3.6** [single solution] A goal \( g \) is single solution for constraint \( c \) and program \( P \) iff the state \( (g, c) \) has at most one successful derivation in \( P \).

**Theorem 3.3.7** If goal \( g_2 \) is both strongly independent of goal \( g_1 \) and single solution for constraint \( c \) and \( P \) then

\[
\#\text{nodes}_P((g_2 : g_1, c)) \leq \#\text{nodes}_P((g_1 : g_2, c)) \]  

The proof comes directly from Lemma 3.2.13 and the given CLP operational semantics. Note that the search space can be decreased for two reasons. First, due to the asymmetry of strong independence \( g_2 \) can decrease the search space of \( g_1 \) for \( c \). Second, the answer for \( g_2 \) (if any) will never be recomputed.

### 3.3.3 Search Independence

As mentioned in Section 3.2.1, within the independent and-parallel model, parallel goals are executed in different environments. The isolation of the environments quite accurately reflects the actual situation in distributed implementations of independent and-parallelism [37]. However, in models designed for shared addressing space machines such isolation of environments is not imposed by the machine architecture and thus, in practice, the goals executing in parallel generally share a single binding environment (e.g. [79, 71, 99]). The amount of overhead introduced by requiring isolated environments (either copying the environment or renaming the goals, plus conjoining the solutions) in these machines, suggests that such isolation should not (unnecessary)

\(^7\)This property is in general undecidable, but can be approximated. Note that the need for this property to hold in the following paragraphs is related to the preservation of the “recomputation” overhead due to the standard backtracking algorithm. Such preservation can also always be ensured by avoiding recomputation through program transformation, encapsulating goals in all-solutions predicates.
be implemented. Furthermore, sharing the environments allow us to avoid performing the conjunction of the answers obtained from the parallel execution, thus reducing the associated overheads.

One could think that if we ensure that $g_2$ is strongly independent of $g_1$ with respect to a given constraint store $c$ then we can execute them in parallel in the same environment while preserving the correctness and efficiency with respect to the sequential execution of $\langle g_1 : g_2, c \rangle$. However, this is not true since, again, we have only considered the successful derivations of $g_1$ and, in this new context, it is then possible that the execution of $\langle g_2, c \rangle$ prunes the search space of $\langle g_1, c \rangle$. For this reason we define a symmetric notion of strong independence. We refer to this concept as search independence:

**Definition 3.3.8** [search independence] Goals $g_1$ and $g_2$ are search independent for constraint $c$ and program $P$ iff

$$\forall c_1 \in \text{partial}_P(\langle g_1, c \rangle) \text{ and } \forall c_r \in \text{partial}_P(\langle g_2, c \rangle) : \text{consistent}(c_1 \land c_r)$$

A collection of goals is search independent for a given $c$ and $P$ iff they are pairwise search independent for $c$ and $P$. Also, a collection of goals is search independent for a set of constraints (interpreted as their disjunction) $C$ and $P$ iff they are search independent for any $c \in C$ and $P$. Finally, a collection of goals is simply search independent for $P$ iff they are search independent for the set of all possible constraints and $P$. ■

Then, in the same spirit as Theorem 3.3.5 we can conclude:

**Corollary 3.3.9** Goals $g_1$ and $g_2$ are search independent for constraint $c$ and program $P$ iff

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

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

### 3.4 Ensuring Search Independence “A Priori”

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

**Definition 3.4.1** [projection independence] Goals $g_1(x)$ and $g_2(y)$ are projection independent for constraint $c$ iff

$$\forall c_1, c_2 \in Cons : \text{if } \exists(c \land \exists_x c_1) \text{ and } \exists(c \land \exists_y c_2) \text{ then } \exists(c \land \exists_x c_1 \land \exists_y c_2) \quad \blacksquare$$

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

**Theorem 3.4.2** Goals $g_1$ and $g_2$ are search independent for constraint $c$ and any program $P$ if they are projection independent for $c$. \quad \blacksquare

Naive application of this sufficient condition implies testing all possible consistent constraints over the variables of each goal. Intuitively, Theorem 3.4.2 holds iff (a) the goals do not have variables in common w.r.t. $c$ and (b) by projecting the constraint store $c$ over the variables of each goal we do not lose “interesting” information for the variables in each goal w.r.t. the original constraint store projected over the variables of both goals, i.e. the former store entails the latter. Therefore, a more useful characterization of projection independence can be captured by:

**Theorem 3.4.3** Goals $g_1(x)$ and $g_2(y)$ are projection independent for constraint $c$ iff

$$(x \cap y \subseteq \text{def}(c)) \text{ and } (\exists_x c \land \exists_y c \rightarrow \exists_{x \cup y} c)^8$$

**Proof** Let us reason by contradiction and assume that there exists $z \in x \cap y$ such that $z \notin \text{def}(c)$. Then, there must exist at least two different values $l_1$ and $l_2$ such that $\exists(c \land z = l_1)$ and $\exists(c \land z = l_2)$ hold. Therefore, $c_1 \equiv \{z = l_1\}$ and $c_2 \equiv \{z = l_2\}$ do not satisfy the conditions required by projection independence, and there is a contradiction. If we assume that $(x \cap y \subseteq \text{def}(c))$ but $(\exists_x c \land \exists_y c) \not\quad \exists_{x \cup y} c$ then, there must exist at least a primitive constraint $c'$ such that $\exists_{x \cup y} c \rightarrow c'$ but $(\exists_{x \cap y} c \land \exists_{y'} c) \not\quad c'$. Therefore, $c'$ must satisfy that $x \cap \text{vars}(c') \not\subseteq \text{def}(c)$, $y \cap \text{vars}(c') \not\subseteq \text{def}(c)$. Let $x'$ and $y'$ be two sequences of variables such that (by an abuse of notation) $x' = x \cap \text{vars}(c')$ and $y' = y \cap \text{vars}(c')$. We can assume without loss of generality that $\text{vars}(c') = x' \cup y'$. Thus, there must exist at least two sequences of values $\bar{x}_x$ and $\bar{y}_y$ such that both $c \land (\bar{x}' = \bar{x}_x)$ and $c \land (\bar{y}' = \bar{y}_y)$ are consistent but $\exists_{x' \cup y'} (c' \land \bar{x}' = \bar{x}_x) \land \bar{y}' = \bar{y}_y$ is not. Thus, $c_1 \equiv x' = \bar{x}_x$ and $c_2 \equiv y' = \bar{y}_y$ do not satisfy the conditions required by projection independence, and there is a contradiction.

---

8Note that $(\exists_x c \land \exists_y c \leftarrow \exists_{x \cup y} c)$ is always satisfied.
The proof for the other direction of the implication is as follows. Assume that the condition holds but there exist two constraints \( c_1 \) and \( c_2 \) such that both \( c \land \exists_{\bar{x}} c \) and \( c \land \exists_{\bar{y}} c_2 \) are consistent but \( c \land \exists_{\bar{x}} c_1 \land \exists_{\bar{y}} c_2 \) is not. By assumption \( \bar{x} \cap \bar{y} \subseteq \text{def}(c) \) and therefore, \( \exists_{\bar{x}} c_1 \land \exists_{\bar{y}} c_2 \) must be consistent. Also by assumption, \( \exists_{\bar{x}} c \land \exists_{\bar{y}} c_2 \) is equivalent to \( \exists_{\bar{x} \cup \bar{y}} c \), therefore, \( \exists_{\bar{x} \cup \bar{y}} c_1 \land \exists_{\bar{x} \cup \bar{y}} c_2 \) must be also consistent, which is in contradiction with \( c \land \exists_{\bar{x}} c_1 \land \exists_{\bar{y}} c_2 \) being inconsistent. ■

**Corollary 3.4.4** Goals \( g_1(\bar{x}) \) and \( g_2(\bar{y}) \) are search independent for constraint \( c \) and any program \( P \) if \( \bar{x} \cap \bar{y} \subseteq \text{def}(c) \) and \( \exists_{\bar{x}} c \land \exists_{\bar{y}} c \rightarrow \exists_{\bar{x} \cup \bar{y}} c \) ■

The proof comes directly from Theorem 3.4.2 and Theorem 3.4.3.

**Example 3.4.1** Consider the goals \( g_1(y), g_2(z) \) and constraint \( c \equiv \{ y > x, z > x \} \). Now \( \exists_{\{y\}} c = \epsilon, \exists_{\{z\}} c = \epsilon, \exists_{\{y,z\}} c = \epsilon \). Therefore, from Corollary 3.4.4, we know that \( g_1(y), g_2(z) \) are search independent for \( c \). ■

The following theorem provides the link with the sufficient conditions defined in [73] (briefly summarized in Section 3.2.1, Definition 3.2.1), for the LP framework.

**Theorem 3.4.5** In the LP framework, two goals are projection independent for constraint \( c \) iff they are strictly independent for \( c \).

**Proof:** Since there is a natural bijection between substitutions and sets of equations in solved form, we let \( c \equiv \text{eqn}(\theta) \) denote the set of equations corresponding to the substitution \( \theta \). Then we can say that two goals \( g_1(\bar{x}) \) and \( g_1(\bar{x}) \) are strictly independent for \( c \) iff \( \bigcup_{z_1 \in \bar{x}, z_1 = t_1 \in c} \text{vars}(t_1) \cap \bigcup_{z_2 \in \bar{y}, z_2 = t_2 \in c} \text{vars}(t_2) = \emptyset \). Let us assume that the goals are strictly independent for \( c \) and reason by contradiction. If there exists a \( w \in (\bar{x} \cap \bar{y}) \) such that \( w \notin \text{def}(c) \), then there must exist an equation \( w = t \) in \( c \) such that \( \text{vars}(t) \neq \emptyset \). Since \( \text{vars}(t) \) is in the intersection of both sets above, the goals are not strict independent, and there is a contradiction. If we assume that \( \bar{x} \cap \bar{y} \subseteq \text{def}(c) \) but \( \exists_{\bar{x}} c \land \exists_{\bar{y}} c \neq \exists_{\bar{x} \cup \bar{y}} c \), as before there must exist a primitive constraint \( d \) entailed by \( c \) such that \( \bar{x} \cap \text{vars}(d) \not\subseteq \text{def}(c) \), and \( \bar{y} \cap \text{vars}(d) \not\subseteq \text{def}(c) \). In the LP framework this only can happen if there exist two equations \( z_1 = t_1 \) and \( z_2 = t_2 \) in \( c \) such that \( z_1 \in \bar{x} \), \( z_2 \in \bar{y} \) and \( \text{vars}(t_1) \cap \text{vars}(t_2) = \emptyset \), which is in contradiction with the assumption of strict independence. The proof for the other direction of the implication follows the same reasoning. ■

As a result, in the Herbrand domain there is no need to actually project the constraint store over any set of variables, thus – due to the characteristics of the implementation of the terms and the operation of the unification algorithm – transforming
a possibly exponential operation into a linear one. However, when constraint domains other than Herbrand are involved, the cost of performing a precise projection may be too high. A pragmatic solution is to find if variables are ‘linked” through the primitive constraints in the constraint store. In fact we can do better by noticing that we can ignore variables that are constrained to take a unique value. In the following we will use Π to denote a sequence of constraints.

More formally, the relation \( \text{link} \) holds for variables \( x \) and \( y \) if there is a primitive constraint \( c \) in \( \Pi \) such that \( \{x, y\} \subseteq \text{vars}(c) \setminus \text{def}(\Pi) \). The relation \( \text{link}_\Pi(x, y) \) is the transitive closure of \( \text{link}_\Pi(x, y) \). We lift \( \text{links} \) to sets of variables by defining \( \text{Links}_\Pi(\vec{x}, \vec{y}) \) iff \( \exists x \in \vec{x} \) and \( \exists y \in \vec{y} \) such that \( \text{links}_\Pi(x, y) \).

**Theorem 3.4.6** Goals \( g_1(\vec{x}) \) and \( g_2(\vec{y}) \) are projection independent for constraint \( \Pi \) if \( \neg \text{Links}_\Pi(\vec{x}, \vec{y}) \). ■

Note that the theorem does not depend on the syntactic representation we choose for \( \Pi \). In fact if the solver keeps a “normal form” for the current constraints we are better off using the normal form rather than the original sequence of constraints as this allows the definition to be simplified. More precisely: constraints \( \Pi \) are in normal form if they have form:

\[
x_1 = f_1(\vec{y}) \land x_2 = f_2(\vec{y}) \land ... \land x_n = f_n(\vec{y}) \land \Pi'
\]

where the \( x_i \) are distinct and disjoint from the variables \( \vec{y} \) and \( \text{vars}(\Pi') \subseteq \vec{y} \). Associated with the normal form is an assignment \( \psi \) to the eliminated variables, namely,

\[
[x_1 \mapsto f_1(\vec{y}), ... x_n \mapsto f_n(\vec{y})]
\]

It is straightforward to verify that \( \text{Links}_\Pi(x, y) \) iff \( \text{Links}_\Pi(\text{vars}(\psi(x)), \text{vars}(\psi(y))) \).

The condition imposed by Theorem 3.4.6, although clearly sufficient, is somewhat conservative. For instance, although the goals \( g_1(y), g_2(z) \) are search independent for \( \Pi = \{y > x, z > x\} \), \( \text{Links}_\Pi(\{y\}, \{z\}) \) holds due to the transitive closure performed when computing \( \text{link}_\Pi(y, z) \). Thus, if projection may be efficiently performed for the particular constraint domain and solver it is better to use Theorem 3.4.3 to determine search independence at run time.

It is interesting to note that despite the fact that we initially considered a left-to-right execution rule, the sufficient conditions given in this section are valid independently of any computation rule. This is due to fact that these conditions are defined in terms of the information provided by the constraint store readily available before executing the goals. Thus, the conditions will remain valid no matter which computation

56
rule will be later applied in the execution of the goals. Therefore, the results obtained in this section can be directly applied to non-deterministic CLP languages with other computation rules, such as AKL [90] or non-deterministic concurrent constraint languages in general [138].

### 3.5 Solver Independence

From the results in previous sections, it may be thought that search space independence is enough for ensuring not only the correctness but also the efficiency of any transformation applied to the search independent goals. Unfortunately, as mentioned in Section 3.3, this is not true in general.

**Example 3.5.1** Consider an empty constraint store, a sequence $\Pi_1$ of primitive constraints which result in a solvable system of linear equations, and another sequence $\Pi_2$ of primitive constraints, each of them uniquely constraining one of the variables involved in the system of equations, where $\Pi_2$ is consistent with $\Pi_1$. It is clear that processing $\Pi_1$ and then $\Pi_2$ will take longer than processing $\Pi_2$ first and then $\Pi_1$. The reason is that while in the former case a relatively complex constraint solving algorithm (such as Gaussian elimination) has to be applied, in the latter only simple value propagation is performed.

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

Therefore, although as shown by the results in previous sections search space preservation ensures that for each transition rule the number of applications of this transition rule in the derivation trees of each state is preserved, it does not ensure the preservation of the amount of work when the $\rightarrow_c$ and $\rightarrow_{cf}$ transition rules are applied and the store.
in each state is different. The main problem is that the constraint solver is viewed as
a black box, i.e., the operational semantics allow us to see the transitions applied at
the higher level, but not those performed by the constraint solver at each of those high
level transitions. If we could have access to such “low level” transitions, the amount
of work performed by the constraint solver in adding a particular constraint to a par-
ticular store, would become explicit, and it could be characterized in terms of search
space, analogously as for the high level transitions. This is in fact the approach taken
by Bueno et al in [18].

Therefore, it is clear that modifying the order in which a sequence of primitive
constraints is added to the store may have a critical influence on the time spent by
the constraint solver algorithm in obtaining the answer, even if the resulting constraint
is consistent. In fact, this issue is the core of the reordering application described in
[110]. The variance of the cost of adding primitive constraints to the store has been
ignored as a factor of negligible influence in traditional logic programming. This is due
to the specific characteristics of the standard unification algorithms [130, 114] – we will
return to this point later. However, as shown before, it cannot be ignored in the context
of CLP languages. For this reason, we now introduce the notion of constraint solver
independence, a new type of independence which, although orthogonal to search space
independence, is also needed in order to ensure the efficiency of several optimizations.

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

\[
\text{cost}(\text{Solv}, c, c”) \leq \text{cost}(\text{Solv}, c, c’) \leq K
\]

We let merge(c, c’) denote the set of all mergings of the constraint sequences c and
c’.

**Definition 3.5.1** [sequence-solver K-independence] Constraint sequences c and c” are
K-independent for store c and solver Solv iff \(\exists (c’ \land c” \land c)\) implies that for every
c_1, c_2 ∈ merge(c’, c”): \(\text{cost}(\text{Solv}, c, c_1) - \text{cost}(\text{Solv}, c, c_2) \leq K\). ■

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

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

**Definition 3.5.2** [solver independence] A constraint solver \( \text{Solv} \) is independent iff for all constraint sequences \( c, c' \) and \( c'' \): \( c' \) and \( c'' \) are K-independent for \( c \) and \( \text{Solv} \), where K is a “small” constant value.

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

**Definition 3.5.3** [projection solver independence] A constraint solver \( \text{Solv} \) is projection independent iff for all constraint sequences \( c, c' \), and \( c'' \), then if \( c' \) and \( c'' \) are projection independent for constraint \( c, c' \) and \( c'' \) are K-independent for \( c \) and \( \text{Solv} \), where K is a “small” constant value.

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

**Definition 3.5.4** [link solver independence] A constraint solver \( \text{Solv} \) is link independent iff for all constraint sequences \( c, c' \) and \( c'' \), if \( \neg \text{links}_c(\text{vars}(c'), \text{vars}(c'')) \) then \( c' \) and \( c'' \) are solver K-independent for \( c \) and \( \text{Solv} \), where K is a “small” constant value.

We claim that most reasonable constraint solvers are link independent and perhaps some are projection independent and that, therefore, the efficiency of many optimizations, such as and-parallelism, can be ensured once the adequate a priori notion is proved to hold for the goals involved in the optimization.

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

In many CLP systems, for example CLP(\( \mathbb{R} \)) [88] and Prolog-III [35], constraint testing over systems of linear equations and inequations is performed using an incremental version of the Simplex algorithm. Essentially this involves incrementally recomputing a normal form for the constraint solver when a new constraint is added. This is done by a succession of “pivots” which exchange the variables being eliminated. When a constraint is first encountered it is “simplified” by eliminating the variables from it. If this reduces the constraint to a simple assignment or Boolean test, then, for efficiency,
the constraint is not passed to the constraint solver but is handled by the constraint “interface”. In order to recognize such assignments or tests the solver keeps track of all variables which are constrained to a unique value. Let this constraint solver be called Simplex. It is easy to construct examples showing that Simplex is neither independent nor projection independent. However, we do have that Simplex is link independent.

We believe that the reason Simplex is link independent is typical of many real solvers. It is instructive to reconsider unification algorithms as solvers for equality constraints over the domain of Herbrand terms and study their independence characteristics. It is clear that most reasonable unification algorithms would satisfy the conditions of projection independence, and in particular those which are “linear”, i.e. which have the property of performing a number of atomic steps which is linear in the size of the terms being unified [130, 114]. Furthermore, if we denote by LinUnif a unification algorithm belonging to the latter class, the we have that: LinUnif is independent.

It is interesting to point out that independence does not hold even within Herbrand for all solvers. For example, the cost of the original unification algorithm of Robinson [132], which is exponential in the worst case, can vary by more than a constant factor depending on reordering. The algorithm used in most practical LP systems is actually an adaptation of Robinson’s. However these algorithms can actually be linear because either they (incorrectly) do not perform the occur check or they simply allow regular trees as well as terms, and also because they do not materialize the substitutions, but rather keep them in an implicit representation using pointers. In fact, in most practical implementations the difference of execution time after reordering will actually be very close to zero. This is the assumption that is used in practice in optimizations of logic programs based on independence and it is this assumption which makes the classical view of expressing independence in LP in terms only of search independence correct.

3.6 Applications

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

Let \( \text{cons}(d) \) denote the sequence of constraints in a derivation \( d \). It follows from our results that in the CLP context a sequence of goals \( g_1 : \cdots : g_n \) can be allowed to (independently) be executed in parallel in the context of \( c \) if\(^9\):

- the goals are strongly independent for \( c \), and
- for every two derivations \( d_i \) and \( d_j \) of \( (g_i, \Pi) \) and \( (g_j, \Pi) \), \( i \neq j \) respectively, \( \text{cons}(d_i), \text{cons}(d_j) \) are K-independent for the solver and for each possible \( \Pi \), where K should somehow be less than the advantage gained by having more than one agent working in parallel (which is clearly the case if K is a small constant).

As mentioned before, given the potential overhead of creating independent environments, however, and given that strong independence is already needed for ensuring correctness, it may be advantageous in practice to require a little more – search independence – since then execution in independent environments is not required. Furthermore, if our parallel system detects that a set of goals is search independent based on the sufficient conditions provided by Theorem 3.4.3 or Theorem 3.4.6 then we can ensure that given a weakly independent solver, efficiency is ensured. The reason is that if one of those theorems holds for those goals, then we can ensure that all constraint sequences generated by different goals will not share variables, unless they are uniquely defined. Therefore we can ensure the second condition mentioned above, i.e., that for all constraint sequences generated by different goals and which are consistent with the store, those constraint sequences are K-independent for the store and the solver, where K is a small constant. In other words, for “reasonable” solvers, the a priori conditions proposed in Section 3.4 are sufficient for efficiency to be ensured, since both search space preservation and either projection or solver independence hold.

CLP languages can take advantage of parallelism from many points of view. First, it is usual in CLP programs to start by establishing a high number of constraints which are always consistent and this can be often done in parallel. Second, complex functions which are used to compute values can also get advantage from their parallel execution. Third, satisfaction of constraints involving terms which do not belong to the same constraint system (e.g., booleans and numbers) are in general obviously independent and can be executed in parallel (unless type conversion is being performed). Finally, since almost all CLP programs subsume the Herbrand domain, CLP languages can take

\(^9\)We also assume the conditions on the scheduling of processors and tasks, and on backtracking given in [75] to ensure that the no slow-down property also holds when a goal (which is not the rightmost) has no answers.
Advantage of all the conventional applications of and-parallelism, i.e. parallel execution of tasks which can be decomposed into subtasks.

As an example, consider the following CLP program:

```
add_matrix([ ], [ ], [ ]).
add_matrix([x|xs], [y|ys], [z|zs]) ←
    add_vector(x, y, z), add_matrix(xs, ys, zs).

add_vector([ ], [ ], [ ]).
add_vector([x|xs], [y|ys], [z|zs]) ←
    z = x + y, add_vector(xs, ys, zs).
```

which performs matrix addition. Usually, `add_matrix` is called with the two first arguments constrained to a unique value (two matrices), the third being an unconstrained variable. Then, the two subgoals in the second clause of both `add_matrix` and `add_vector` can be run in parallel, significantly improving the performance. Furthermore, this parallel execution is also allowed whenever one argument is uniquely defined and the other two are unconstrained variables.

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

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

### 3.6.2 Stability Detection

The notion of “stability” [90] is used in the Andorra family of languages in general and in the AKL language in particular as the rule for control of one of the basic operations of the language – global forking. This operation amounts to starting and-parallel execution of a goal which is non-deterministic. Stability for a goal is defined informally as being in a state in which other goals running in parallel with it will not affect its execution. This is of course an undecidable notion and in practice sufficient conditions are used in actual implementations.

In particular, in the first implementation of AKL, restricted to the Herbrand domain, the stability condition used is actually the classical notion of strict independence for LP [58]. Since the AKL language is defined to be a constraint language, the notion of stability has to be generalized to the constraint level. As we have shown, generalization
cannot be done by directly applying naive liftings of the LP concepts of independence. We believe that the results presented in this chapter will be of direct application.

### 3.6.3 Reordering

In [110] an optimization based on reordering the goal $c \land g$ to $g \land c$ where $c$ is a primitive constraint is suggested whenever $c$ and $g$ are strongly independent. The motivation for this is that variables in $c$ may become uniquely defined by $g$, enabling the constraint $c$ to be replaced by either an assignment statement or a simple Boolean test. If this is true, especially in the case $g$ is recursive, large speedups are obtained. We can lift this idea to the level of goals and thus reorder goals as well.

Consider the (sub-)goal $g_1 : g_2$ appearing in some program, and assume that this will be called with the constraint sequences $c_1, c_2, \ldots$. This should be reordered to $g_2 : g_1$ if the following two conditions are met. Firstly that changing the ordering will not increase the search space. From Theorem 3.3.7 a sufficient condition is that $g_2$ is single solution and strongly independent of $g_1$ and $\text{answers}_P(g_1, c_i) \neq \emptyset$ for each $c_i$. Secondly there should be an improvement in the overall execution time. Thus, for each $c_i$ and for each derivation $d_1$ of $\langle g_1, c_i \rangle$ and each derivation $d_2$ of $\langle g_2, c_i \rangle$, $\text{cost}(\text{Solv}, c_i, \text{cons}(d_2) :: \text{cons}(d_1)) \leq \text{cost}(\text{Solv}, c_i, \text{cons}(d_1) :: \text{cons}(d_2))$ where $\text{Solv}$ is the constraint solver.

As an example, consider the program FIB for computing the Fibonacci numbers.

\[
\begin{align*}
\text{fib}(0,1). & \quad \text{(FIB)} \\
\text{fib}(1,1). \\
\text{fib}(N,F) \leftarrow & \\
& \quad N_1 = N - 1, \; N_2 = N - 2, \; F = F_1 + F_2, \\
& \quad \text{fib}(N_1,F_1), \; \text{fib}(N_2,F_2).
\end{align*}
\]

We consider the (usual) case that this is called with constraints in which the first argument is constrained to an integer and the second argument is an unconstrained variable. In this case both recursive calls to fib are single solution. Furthermore, because $F$, $F_1$, and $F_2$ are initially unconstrained, $\text{fib}(N_1,F_1):\text{fib}(N_2,F_2)$ is strongly independent of $F = F_1 + F_2$ w.r.t. the given store. Thus the recursive clause body can be reordered to give the optimized program:

\[
\begin{align*}
\text{fib}(0,1). & \quad \text{(0-FIB)} \\
\text{fib}(1,1). \\
\text{fib}(N,F) \leftarrow & \\
& \quad N_1 = N - 1, \; N_2 = N - 2, \\
& \quad \text{fib}(N_1,F_1), \; \text{fib}(N_2,F_2), \; F = F_1 + F_2.
\end{align*}
\]

63
The advantage of 0-FIB over FIB is that all of the constraints in 0-FIB are reduced to simple assignments or tests meaning that the (expensive) constraint solver is not called when 0-FIB is executed, giving rise to substantial performance improvement.

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

### 3.6.4 Intelligent Backtracking

Backtracking is the operation performed whenever a state can only be reduced to fail. In this situation, the computation has to return to the state (or at least an equivalent state) from which the last non deterministic reduction was performed (called the last choice point). In Prolog, constraints are equations between terms, internally represented as binding of variables. During forward execution, the variables simply come more and more instantiated. Therefore, backtracking just implies unbinding the variables which have became more instantiated since the last choice point. As a result, the only information to be kept in order to allow a correct backtracking is the set of variables which have become bound.

On the contrary, in CLP a constraint expression can be completely changed from its original form. Therefore, usually we will need to keep track of the constraints in their original forms. Keeping such information is significantly more costly than keeping track of the variables. Thus, developing techniques which avoids a many backtracking step as possible seems to be even more important when CLP languages are taken into account.

Intelligent backtracking [13] consists in analyzing, upon failure, the causes of the failure and determining appropriate backtracking points that can eliminate the failure while maintaining correctness, thus avoiding unnecessary computations. The method used in LP is based both on an extended unification algorithm which keeps track of the history of the unification and performs failure analysis, and a backtrack process, which is essentially the same in all methods. One of the main decisions in this application is related to the accuracy of the unification history representation: an extremely accurate representation could be intractable, a too simple one could perform a naive backtracking at a high cost.

We propose a simple form of intelligent backtracking based on the notion of weak independence. Let \(g_1 : \cdots : g_n\) be a set of goals which are weakly independent for the store \(c\). Theorem 3.3.3 ensures that whenever there exists a goal \(g_i, 1 \leq i \leq n\) for which no answers for goal \(g_i\) are found, execution can safely backtrack to the choice-point placed just before \(g_1\), skipping all the choice-points in between.
It could be thought that, although the time saved by such optimization can be significant, the complexity of the tests needed for a run-time detection of weakly independent goals may yield a slow-down, which is clearly not the aim. However, it is important to note that, first, the traditional techniques applied to LP (i.e. keeping track of the history of the unification), are no longer applicable without modifications when domains other than Herbrand are considered. Second, that maintaining a comparable structure to be able to accurately determine the causes of a failure for, for example, the constraint system based on reals with equalities and inequalities can be very complex. And third, that CLP languages are usually defined over more than one constraint system, which increases significantly the complexity of the problem. Therefore, we believe that inferring independence at compile-time or partly at compile-time and partly at run-time, and providing accurate information to the compiler so that it can specialize the program code in order to provide the appropriate links, can be a quite a useful technique for efficiently implementing intelligent backtracking in the context of CLP languages.

3.7 Chapter Conclusions

We have shown how a simple extrapolation of the LP-based definitions of independence to CLP turns out to be both not general enough in some cases and erroneous in others, and identified the need in CLP for defining concepts of independence both at the search level and at the solver level. Several such concepts have been presented and shown to be relevant to several classes of applications. We have also proposed sufficient conditions for the concepts of independence proposed, which are easier to detect at run-time than the original definitions. Also, it has been shown how the concepts proposed, when applied to conventional LP, render the traditional notions and are thus a strict generalization of such notions. Finally, we believe we have in addition provided some insights into hidden assumptions related to properties of the standard unification algorithms that were made in the development of the LP concepts.

It is our belief that using the concepts of independence presented the range of applications independence-related optimizations can be even larger in CLP than in LP.
There has been widespread interest in a class of second-generation logic programming languages that provide more flexible scheduling than those based on a fixed scheduling rule. In this new class of languages computation generally proceeds following also some fixed scheduling rule but some calls are dynamically “delayed” until their arguments are sufficiently instantiated to allow the call to run efficiently. Such dynamic scheduling overcomes the problems associated with traditional languages and their fixed scheduling. First, it allows the same program to have many different and efficient operational semantics, as the operational behavior depends on which arguments are supplied in the query. Thus, programs behave efficiently as relations, rather than as functions. Second, the treatment of negation is sound, as negative calls are delayed until all arguments are ground. Third, it allows intelligent search in combinatorial constraint problems. Finally, dynamic scheduling allows a style of programming in which procedures are viewed as processes which communicate asynchronously through dependent variables.

Unfortunately, as we will show in this chapter, the notions of independence developed so far are not valid for this class of languages. As discussed in the previous chapter, when preserving the search space in languages with fixed scheduling, problems only appear when a derivation is pruned due to an inconsistency with a constraint previously added to the store by another goal. In languages with dynamic scheduling the search space can also vary due to the awakening or delaying of some literals. Note that the problems posed by dynamic scheduling subsume those posed by passive constraints in constraint logic programming languages. The main difference is that in dynamically scheduled languages not only constraints but also atoms can be delayed until their arguments become sufficiently instantiated.
In this chapter we extend the notion of independence to dynamically scheduled languages. The interest of such extension is threefold. Firstly, due to the benefits mentioned above, most real (constraint) logic languages, such as IC-Prolog [25], MU-Prolog [127], NU-Prolog [147], Prolog-II [56], Sicstus-Prolog [21], Prolog-III [35], CHIP [68], Prolog M, and SEPIA [55], etc., already provide such flexible scheduling. Therefore the generality and practicality of the applications based on the independence notion will not be a reality for these languages until such applications handle dynamically scheduled programs. Secondly, dynamic scheduling has a significant cost; literals, if affected by a delay declaration, must be checked to see whether they should delay or not; upon variable binding, possibly delayed calls must be woken or put in a “pending” list, so that they are woken before the next literal in the goal is executed; also, few register allocation optimizations can be performed for delayed objects; finally, space needs to be allocated for delayed objects until they are woken [20]. These performance problems make these languages good candidates for many optimizations and, in particular, those based on the independence concept. Thirdly, dynamically scheduled languages have been considered as good target languages for the implementation of concurrent constraint logic languages [51, 137, 149]. The reasons for such decision include the semantic similarities and the progress on implementation and optimization techniques for dynamically scheduled languages and, in particular, the benefits provided by the work on, and implementations of, or- and and-parallel systems. However, taking advantage of such and-parallel systems will only be possible if the independence concept is extended to the class of logic languages with dynamic scheduling.

The rest of this chapter proceeds as follows: in the next section we give a simple example which illustrates the usefulness of dynamic scheduling. In Section 4.2 we give the additional notation that will be used throughout this chapter and an operational semantics of (constraint) logic languages with dynamic scheduling. In Section 4.3 we consider independence in the simpler context of CLP languages with passive constraints. In Section 4.4 we consider a more general context in which atoms are also allowed to be delayed, but we still require that the sequence of delayed literals before the parallel execution of two goals be free of atoms. Finally, in Section 4.5 we consider the general case. Section 4.6 gives sufficient conditions (for the general case) that are easier to detect at run-time than the definitions of independence. Section 4.7 briefly discusses the notion of independence at the solver level. Finally, Section 4.8 presents our conclusions.
4.1 Example

The following program adapted from Naish [126], illustrates the power of allowing calls to delay and the information our analysis can provide. The program \texttt{permute} is a simple definition of the relationship that the first argument is a permutation of the second argument. It makes use of the procedure \texttt{delete(x,y,z)} which holds if \(z\) is the list obtained by removing \(x\) from the list \(y\).

\begin{verbatim}
permute(x,y) ← x = nil,  
y = nil.
permute(x,y) ← x = u : x1,  
delete(u,y,z),  
permute(x1,z).

delete(x,y,z) ← y = x : z.
delete(x,y,z) ← y = u : y1,  
z = u : z1,  
delete(x,y1,z1).
\end{verbatim}

Clearly the relation declaratively given by \texttt{permute} is symmetric. Unfortunately, the behavior of the program with traditional Prolog is not: Given the query \(q_1 \equiv \langle \text{permute}(x,a:b:nil),\epsilon \rangle\) (where \(\epsilon\) denotes the empty constraint), Prolog will correctly backtrack through the answers \(x = a:b:nil\) and \(x = b:a:nil\). However, for the query \(q_2 \equiv \langle \text{permute}(a:b:nil,x),\epsilon \rangle\) Prolog will first return the answer \(x = a:b:nil\) and on subsequent backtracking will go into an infinite derivation without returning any more answers.

For languages with delay the program \texttt{permute} does behave symmetrically. For instance, if the above program is given to the NU-Prolog compiler, a pre-processor will generate the following delay declarations (referred to as \texttt{when} declarations):

\begin{verbatim}
?- \texttt{permute}(x,y) \texttt{when} \(x\) or \(y\).
?- \texttt{delete}(x,y:z,u) \texttt{when} \(z\) or \(u\).
\end{verbatim}

These may be read as saying that the call \texttt{permute(x,y)} should delay until \(x\) or \(y\) is not a variable, and that the call \texttt{delete(x,y:z,u)} should delay until \(z\) or \(u\) is not a variable. Of course programmers can also annotate their programs with \texttt{when} declarations. Given these declarations, both of the above queries will behave in a symmetric fashion, backtracking through the possible permutations and then failing.
What happens is that with \( q_1 \) execution proceeds as in standard Prolog because no literals are delayed. With \( q_2 \), however, calls to delete are delayed and only woken after the recursive calls to permute.

### 4.2 Operational Semantics

In this section we give some additional notation and an operational semantics for constraint logic programs with dynamic scheduling. This semantics simply generalizes the operational semantics of logic programs in which atoms can delay and the operational semantics of constraint logic programs in which constraints can delay.

Assume that the syntactic objects and constraint systems are as defined in Section 3. Also assume that clauses in a program are in normalized form. The operational semantics of a program \( P \) can be presented as a transition system on states \( (G,c,D) \) where \( G \) is a multi-set of literals, \( c \) is a constraint, and \( D \) is a multi-set of delayed literals. Analogously to the operational semantics of constraint logic languages, \( D \) is formed by the literals playing a passive role.

The transition system is parameterized by four functions, namely consistent, infer, delay and woken. The functions consistent\((c)\) and infer\((c,s)\) are those defined in Section 3.1. The function delay\((a,c)\) holds if a call to atom \( a \) delays with the constraint \( c \). The function woken\((D,c)\) returns the multi-set of atoms in the sequence of delayed literals \( D \) that are woken by constraint \( c \). Note that the order of the calls returned by woken is system dependent. The transitions in the transition system are:

- \( (G \cup a, c, D) \xrightarrow{d} (G, c, D \cup a) \) if \( a \) is an atom selected by the computation rule and delay\((a,c)\) holds.
- \( (G \cup a, c, D) \xrightarrow{r} (G \cup B, c, D \cup (a = h)) \) if \( a \) is an atom selected by the computation rule, delay\((a,c)\) does not hold, \( r \equiv h \leftarrow B \) is a clause of \( P \) renamed to new variables, and \( h \) and \( a \) have the same predicate symbol.
- \( (G \cup a, c, D) \xrightarrow{r_f} fail \) if \( a \) is an atom selected by the computation rule, delay\((a,c)\) does not hold, and for every clause \( r \equiv h \leftarrow B \) of \( P \), \( h \) and \( a \) have different predicate symbols.
- \( (G, c, D) \xrightarrow{w} (G \cup D', c, D \setminus D') \) if woken\((D,c) = D'\), and \( D' \) is not empty.
- \( (G \cup c', c, D) \xrightarrow{c} (G, c, D \cup c') \) if \( c' \) is a constraint selected by the computation rule.
- \( (G, c, D) \xrightarrow{i} (G, c', (D \setminus s) \cup s') \) if \((c', s') = infer(c, s)\) and \( s \) is the set of constraints in \( D \).
• \( (G, c, D) \rightarrow_t (G, c, D) \) if \( \text{consistent}(c) \).

• \( (G, c, s) \rightarrow_f \text{fail} \) if \( \neg \text{consistent}(c) \).

Note that this operational semantics can be instantiated to a semantics equivalent to that described in Chapter 3.1 if no atom is delayed or woken.

The notions related to derivations, derivation trees, answers, and partial answers can be obtained as straightforward extensions of those defined for the CLP context. Note that, as in the general CLP operational semantics, the answers can be associated with either successful or floundering derivations. As before, we denote the set of answers to state \( s \) for program \( P \) by \( \text{answer}_P(s) \), the partial answers by \( \text{partial}_P(s) \), and the derivation tree by \( \text{tree}_P(s) \).

In order to simplify the definitions, in the rest of this chapter we will not distinguish between the particular constraints returned by the \( \text{in} \text{fer} \) function and an equivalent result, i.e., if \( \text{in} \text{fer}(c_1, D_1) = (c^1, D^1) \), \( \text{in} \text{fer}(c_2, D_2) = (c^2, D^2) \), \( c^1 \leftrightarrow c^2 \) and \( D^1 \leftrightarrow D^2 \), we will say that \( \text{in} \text{fer}(c_1, D_1) = \text{in} \text{fer}(c_2, D_2) \), \( \text{in} \text{fer}(c_1, D_1) = (c^2, D^2) \) and \( \text{in} \text{fer}(c_2, D_2) = (c^1, D^1) \).

### 4.3 First Case: Constraint Logic Programs with Passive Constraints

In Section 3.2.2 we defined the notion of search space preservation and presented the basic properties which allow us to establish the relationship between this notion and that of independence, in the context of CLP languages. In this section we will discuss the problems posed by dynamically scheduled languages for the preservation of search space and the relationship between search space preservation and independence in this new context. In order to simplify the discussion we will consider three cases. In this section, we consider the case in which only constraints are allowed to be dynamically scheduled, and thus atoms will behave as in previous chapters. In the next section, we will relax this condition by allowing atoms to be dynamically scheduled but we will only consider the problem of search space preservation in the case of states in which the sequence of delayed literals does not contain atoms. Finally, we will completely relax such conditions allowing any kind of delay behavior.

Since in this section the multi-set of delayed literals only contains constraints, by an abuse of notation, in the rest of this section we will consider it as the constraint formed by the conjunction of its elements. In order to start from a particular sequential semantics with which to establish the appropriate comparisons, we will particularize the operational semantics described in the previous chapter. Firstly, the computation rule
implies left-to-right execution order. Thus, the element $G$ of a state $s = (G, c, D)$ will be represented by a sequence of literals instead of by a multi-set. Secondly, the operational semantics can be described by $\rightarrow_{rf}$, $\rightarrow_{rit}$, $\rightarrow_{cit}$, and $\rightarrow_{citf}$ transitions. This implies that the underlying CLP system is quick-checking and progressive. We will consider such transitions as the basic transition rules. This allows us to consider transition rules which are pairwise exclusive, thus greatly simplifying the discussion. Note that, since we have assumed that only constraints are dynamically handled, transitions $\rightarrow_d$ and $\rightarrow_w$ are not allowed. Also, note that, given the basic transition rules proposed, for every state $(G, c, D)$ in the derivation of another state $s$: $\text{infer}(c, D) = (c, D)$. We will extend this property to the queries, i.e., we will assume that for every state $(G, c, D)$ considered: $\text{infer}(c, D) = (c, D)$.

Finally, we will assume that the conditions imposed in the previous chapter for consistent hold and we will require similar conditions from $\text{infer}$. In particular, it should not take variable names into account:

Let $\rho \in \text{Ren}$. $\rho(\text{infer}(c, D)) = \text{infer}(\rho(c), \rho(D))$

It should be idempotent

If $(c_1, D_1) = \text{infer}(c, D)$ then $(c_1, D_1) = \text{infer}(c_1, D_1)$

and,

If $c \leftrightarrow c_1 \land c_2$ then $\text{infer}(c, D) = \text{infer}(c_1, D \land c_2)$

### 4.3.1 Independence and Search Space Preservation

Let us redefine the and-parallel execution model in this new context. Assume that, given the state $(g_1 : g_2 : G, c, D)$ in which $D$ is a multi-set of constraints, and the program $P$ in which no atoms are allowed to delay, we want to execute $g_1$ and $g_2$ in parallel (the extension to more than two goals is straightforward). Then the execution scheme is the following:

- execute $(g_1, c, D)$ and $(g_2, c, D)$ in parallel (in different environments) obtaining the answer constraints $(c_1, D_1)$ and $(c_r, D_r)$ respectively;
- obtain $(c_s, D_s) = \text{infer}(c_1 \land c_r, D_1 \land D_r)$;
- execute $(G, c_s, D_s)$.

As before, and in order to avoid problems when obtaining $c_s$ and $D_s$, it is assumed that the new variables introduced during the renaming steps in the parallel execution of the goals belong to disjoint sets.

Let us now redefine correctness and efficiency of the model.
**Definition 4.3.1** Let \(\langle g_1 : g_2 : G, c, D \rangle\) be a state in which \(D\) is a multi-set of constraints and \(P\) be a program in which no atoms are allowed to delay. The parallel execution of \(g_1\) and \(g_2\) is correct iff for every \(\langle c_1, D_1 \rangle \in \text{answers}_P(\langle g_1, c, D \rangle)\) there exists a renaming \(\rho \in \text{Ren}\) and a bijection which assigns to each \(\langle c_s, D_s \rangle \in \text{answers}_P(\langle g_2, c_1, D_1 \rangle)\) a \(\langle c_r, D_r \rangle \in \text{answers}_P(\langle g_2, c, D \rangle)\)\(^1\) with \((c_s, D_s) = \inf(c_1 \land \rho(c_1), D_1 \land \rho(D_1))\). ■

As before, we assume that the cost of conjoining the answers, i.e., of obtaining \((c_s, D_s)\), is negligible and that the application of a particular transition rule has the same cost (always greater than zero), independently of the state to which the transition is applied. Let \(TR\) be the set of different transition rules that can be applied. Let \(s\) be a state and \(N(i, s)\) be the number of times in which a particular transition rule \(i \in TR\) has been applied in \(tree_p(s)\). Let \(K(i)\) be the cost of applying a particular transition rule \(i \in TR\).

**Definition 4.3.2** Let \(\langle g_1 : g_2 : G, c, D \rangle\) be a state in which \(D\) is a multi-set of constraints and \(P\) be a program in which no atoms are allowed to delay. The parallel execution of \(g_1\) and \(g_2\) is efficient iff for every \(\langle c_1, D_1 \rangle \in \text{answers}_P(\langle g_1, c, D \rangle)\):
\[
\sum_{i \in TR} K(i) \ast N(i, \langle g_2, c, D \rangle) \leq \sum_{i \in TR} K(i) \ast N(i, \langle g_2, c_1, D_1 \rangle).
\]

Since the definition of search space preservation is identical to Definition 3.2.6, let us focus on the properties of search space preservation and the relationship of search space preservation with the preservation of correctness and efficiency of the parallel execution with respect to the sequential execution.

The first point to be noticed is that a delayed (or passive) constraint can become active or remain delayed without affecting the search space of the subsequent branches in the derivation tree.

**Example 4.3.1** Consider the state \(\langle p(x, y) : q(y), \epsilon, x \ast w = v \rangle\) and the program:

\[
\begin{align*}
p(x, y) & \leftarrow x = y. \\
qu(y) & \leftarrow y = 4, s(y, z). \\
s(y, z) & \leftarrow z > y. \\
s(y, z) & \leftarrow z < y.
\end{align*}
\]

where no atom can be delayed. The only answer obtained for \(\langle p(x, y), \epsilon, x \ast w = v \rangle\) is \(\langle x = y, x \ast w = v \rangle\). Figure 4.1 shows the derivation tree for states \(\langle q(y), x = y, x \ast w = v \rangle\) and \(\langle q(y), \epsilon, x \ast w = v \rangle\). Note that the renaming steps have been avoided for simplicity.

\(^1\)As before, the suffixes _s_ and _r_ will be associated to the arguments of the states obtained during the sequential execution and the parallel execution of the goal to the right, respectively.

73
It is clear that although the delayed constraint $y \times x = w$ becomes active at different points of the execution, the derivation trees for $\langle q(y), x = y, x \times w = v \rangle$ and $\langle q(y), \epsilon, x \times w = v \rangle$, are the same. □

![Diagram](image.png)

**Figure 4.1:**

**Example 4.3.2** Consider the state $\langle p(x) : q(x, y), \epsilon, nil \rangle$ and the program:

\[
\begin{align*}
p(x) & \leftarrow x = 4. \\
q(x, y) & \leftarrow y \times x = w, s(x, w). \\
s(x, w) & \leftarrow x > w. \\
s(x, w) & \leftarrow x < w.
\end{align*}
\]

where no atom can be delayed. The only answer obtained for $\langle p(x, y), \epsilon, nil \rangle$ is $\langle x = 4, nil \rangle$. It is easy to see that although the constraint $y \times x = w$ becomes delayed in both branches of the the derivation tree for state $\langle q(y), \epsilon, nil \rangle$ and does not become delayed in any branch of the the derivation tree for state $\langle q(y), x = 4, nil \rangle$, the derivation trees of those two states are the same. □

The situation is similar to the one considered in the previous chapter since, again, the search space can only be affected by pruning some branches. Therefore, we can still state that search space preservation is sufficient for ensuring correctness:

**Theorem 4.3.3** Let $\langle g_1 : g_2 : G, c, D \rangle$ be a state in which $D$ is a multi-set of constraints and $P$ be a program in which no atom can be delayed. The parallel execution of $g_1$ and $g_2$ is correct if for every $\langle c_1, D_1 \rangle \in answers_P(\langle g_1, c, D \rangle)$, the search spaces of $\langle g_2, c, D \rangle$ and $\langle g_2, c_1, D_1 \rangle$ are the same for $P$. ■

As in the previous chapter, the proof comes from the following two lemmas, which are extensions of Lemma 3.2.8 and Lemma 3.2.9, respectively.
Lemma 4.3.4 Let \( \langle g_2, c_1, D_1 \rangle \) and \( \langle g_2, c, D \rangle \) be two states with an identical sequence of literals in which both \( D_1 \) and \( D \) are multi-sets of literals, and \( P \) be a program in which no atom can delay. There exists a renaming \( \rho \in Ren \) such that for every two non-failure nodes \( s \equiv \langle G_s, c_s, D_s \rangle \) and \( r \equiv \langle G_r, c_r, D_r \rangle \) with the same path in \( \text{treep}(\langle g_2, c_1, D_1 \rangle) \) and \( \rho(\text{treep}(\langle g_2, c, D \rangle)) \), respectively: \( G_s \equiv G_r \).

The proof is identical to that performed for Lemma 3.2.8, based on the operational semantics. Therefore, as before, we can ensure that, as constructed, \( \rho(\text{treep}(\langle g_1, c_1, D_1 \rangle)) \equiv \text{treep}(\langle g_1, c_1, D_1 \rangle) \) and \( \rho(\langle g_2, c, D \rangle) \equiv \langle g_2, c, D \rangle \). In the rest of this section we assume that \( \rho \) satisfies such conditions.

Lemma 4.3.5 Let \( \langle g_2, c_1, D_1 \rangle \) and \( \langle g_2, c, D \rangle \) be two states with an identical sequence of literals in which both \( D \) and \( D_1 \) are multi-sets of constraints, \( c_1 \rightarrow c \) and \( c_1 \land D_1 \rightarrow D \). Let \( P \) be a program in which no atom is allowed to delay and \( \rho \) be the renaming from Lemma 4.3.4. Then, for every two nodes \( s \equiv \langle G_s, c_s, D_s \rangle \) and \( r \equiv \langle G_r, c_r, D_r \rangle \) with the same path in \( \text{treep}(\langle g_2, c_1, D_1 \rangle) \) and \( \rho(\text{treep}(\langle g_2, c, D \rangle)) \), respectively: \( (c_s, D_s) = \text{infer}(c \land c_r, D_1 \land D_r) \).

**Proof:** Let us reason by induction. It is clear that the conditions are satisfied for the base case: since \( c_1 \rightarrow c \) and \( c_1 \land D_1 \rightarrow D \), we have that \( \text{infer}(c_1 \land c, D_1 \land D) = \text{infer}(c_1, D_1) \). Consider now that the conditions are satisfied for \( s' \equiv \langle G'_s, c'_s, D'_s \rangle \) and \( r' \equiv \langle G'_r, c'_r, D'_r \rangle \). If the leftmost literal in \( s' \) (and thus in \( r' \)) is a constraint \( c' \), then the nodes \( s \equiv \langle G_s, c_s, D_s \rangle \) and \( r \equiv \langle G_r, c_r, D_r \rangle \) obtained from \( s' \) and \( r' \) respectively will be as follows. By Lemma 4.3.4 \( G_s \equiv G_r \). By definition of the operational semantics \( (c_s, D_s) = \text{infer}(c'_s, c' \land D'_s) \). By assumption \( (c'_s, D'_s) = \text{infer}(c_1 \land c'_1, D_1 \land D'_1) \), and thus \( (c_s, D_s) = \text{infer}(c_1 \land c'_1, c' \land D_1 \land D'_1) \). By definition of the operational semantics, \( (c_r, D_r) = \text{infer}(c'_r, c' \land D'_r) \) and thus \( (c_s, D_s) = \text{infer}(c_1 \land c'_1, c' \land D_1 \land D'_r) \). If the leftmost literal in \( s' \) (and thus in \( r' \)) is an atom \( a' \), since \( s \) and \( r \) must be obtained using the same rule renamed to the same variables \( r : h \leftarrow B \), the reasoning is similar with the constraint being \( a = h \).

Note that, as pointed out in the previous chapter, even if the consistent function associated with the underlying constraint system is not complete, and thus \( c_s \) possibly entails fail, the theorem is satisfied.

Those two lemmas, and the fact that search space preservation implies a bijection among answers, allow us to prove that search space preservation is sufficient for ensuring the correctness of the parallel execution. Ensuring that the efficiency is also guaranteed is even easier due to the definition of search space.
Theorem 4.3.6 Let \( \langle g_1 : g_2 : G, c, D \rangle \) be a state in which \( D \) is a multi-set of constraints, \( P \) a program in which no atom is allowed to delay, and \( \langle c_1, D_1 \rangle \in \text{answers}_P(\langle g_1, c, D \rangle) \). If the search spaces of \( \langle g_2, c, D \rangle \) and \( \langle g_2, c_1, D_1 \rangle \) are the same for \( P \), then:

\[
\sum_{i \in TR} K(i) \ast N(i, \langle g_2, c, D \rangle) = \sum_{i \in TR} K(i) \ast N(i, \langle g_2, c_1, D_1 \rangle).
\]

The proof is identical to that of Theorem 3.2.10, and thus we can ensure the following:

Theorem 4.3.7 Let \( \langle g_1 : g_2 : G, c, D \rangle \) be a state in which \( D \) is a multi-set of constraints, and \( P \) be a program in which no atoms are allowed to delay. The parallel execution of \( g_1 \) and \( g_2 \) is efficient if for every \( \langle c_1, D_1 \rangle \in \text{answers}_P(\langle g_1, c, D \rangle) \) the search spaces of \( \langle g_2, c, D \rangle \) and \( \langle g_2, c_1, D_1 \rangle \) are the same for \( P \).

We have now proved that search space preservation is sufficient for ensuring the correctness and also for ensuring the efficiency of the parallel execution. However, analogously to the case in which no delayed constraints were allowed, we can go further and show that it is in fact necessary for having both correctness and efficiency. The following lemmas, extensions of Lemma 3.2.12 and Lemma 3.2.13, are instrumental for this result.

Lemma 4.3.8 Let \( \langle g_2, c_1, D_1 \rangle \) and \( \langle g_2, c, D \rangle \) be two states with an identical sequence of literals in which both \( D \) and \( D_1 \) are multi-set of constraints, \( c_1 \rightarrow c \) and \( c_1 \land D_1 \rightarrow D \). Let \( P \) be a program in which no atom is allowed to delay. For every two nodes \( s \) and \( r \) with the same path in \( \text{tree}_P(\langle g_2, c_1, D_1 \rangle) \) and \( \text{tree}_P(\langle g_2, c, D \rangle) \), respectively: \( s \) and \( r \) have been obtained with the same transition rule iff either \( s \equiv r \equiv \text{fail} \) or they are both non failure nodes.

The proof is identical to that of Lemma 3.2.12, based on Lemma 4.3.4 and the definition of the operational semantics, once the application rules are changed appropriately, i.e., replacing \( \rightarrow_{\text{r}}, \rightarrow_{\text{c}}, \rightarrow_{\text{rf}}, \) and \( \rightarrow_{\text{cf}} \) with \( \rightarrow_{\text{rf}}, \rightarrow_{\text{cf}}, \rightarrow_{\text{rf}}, \) and \( \rightarrow_{\text{cf}} \), respectively.

Lemma 4.3.9 Let \( P \) be a program in which no atom is allowed to delay. Let \( \langle g_2, c_1, D_1 \rangle \) and \( \langle g_2, c, D \rangle \) be two states with an identical sequence of literals in which both \( D \) and \( D_1 \) are multi-set of constraints, \( c_1 \rightarrow c, c_1 \land D_1 \rightarrow D \) and the search spaces of \( \langle g_2, c_1, D_1 \rangle \) and \( \langle g_2, c, D \rangle \) are different for \( P \). Then, there exists a bijection which assigns to each node \( s \) in \( \text{tree}_P(\langle g_2, c_1, D_1 \rangle) \) for which there is no corresponding node in \( \text{tree}_P(\langle g_2, c, D \rangle) \), a node \( r \) in \( \text{tree}_P(\langle g_2, c, D \rangle) \) with the same path, such that \( s \) and \( r \) have been obtained applying the \( \rightarrow_{\text{cf}} \) and \( \rightarrow_{\text{cf}} \) transition rule, respectively, and the parents of \( s \) and \( r \) correspond.
The proof is also almost identical to that of Lemma 3.2.13 when using the results provided by the above lemmas. As a result we can state the following:

**Theorem 4.3.10** Let \( g_1 : g_2 : G, c, D \) be a state in which \( D \) is a multi-set of constraints, and \( P \) be a program in which no atoms are allowed to delay. The parallel execution of \( g_1 \) and \( g_2 \) is correct and efficient iff for every \( \langle c_1, D_1 \rangle \in \text{answers}_P(g_1, c) \) the search spaces of \( \langle g_2, c, D \rangle \) and \( \langle g_2, c_1, D_1 \rangle \) are the same for \( P \).

Again, the proof is identical to that of Theorem 3.2.14, when changing the transition rules appropriately. Those results allow us to extend in a straightforward way the corollaries obtained regarding the preservation of the number of non-failure nodes and the success of the back-bindings.

Therefore, following the definitions of correctness and efficiency given above, we can ensure that in the restricted context considered, search space preservation ensures both the correctness and efficiency of the parallel execution of goals. Furthermore, that search space preservation is not only a sufficient but also a necessary condition for ensuring that both efficiency and correctness hold.

We now extend the notions of independence presented in Section 3.3 to this new context, based on the previous results.

### 4.3.2 Weak Independence

As mentioned before, weak independence aims at characterizing those goals for which the particular form of intelligent backtracking described in Section 3.6.4 can be safely performed. Thus, we are just interested in a characterization for which the conditions stated in Theorem 3.3.3 (suitably changed for dynamically scheduled languages) hold.

I.e., that given a collection of goals \( g_1 : \cdots : g_n \), constraint \( c \), multi-set of constraints \( D \) and program \( P \), if there exists \( \langle c_1, D_1 \rangle \in \text{answers}_P(g_1 : \cdots : g_{i\ldots n}, c, D) \) with \( \text{answers}_P(g_i, c_1, D_1) = \emptyset \), then, for every \( \langle c_2, D_2 \rangle \in \text{answers}_P(g_1 : \cdots : g_{i\ldots n}, c, D) : \text{answers}_P(g_i, c_2, D_2) = \emptyset \). As in Section 3.3.1, this characterization is related to the preservation of search space in the finite, non-failure branches of the derivation tree, and thus can be defined as follows:

**Definition 4.3.11** [Weak independence] Goals \( g_1 \) and \( g_2 \) are weakly independent for constraint \( c \), multi-set of delayed constraints \( D \), and program \( P \) in which no atoms can be delayed iff

\[
\forall \langle c_1, D_1 \rangle \in \text{answer}_P(g_1, c, D) \quad \text{and} \quad \forall \langle c_r, D_r \rangle \in \text{answer}_P(g_2, c, D) : \text{consistent}(c) 
\]

where \( c \) is the active constraint obtained by \( \text{infer}(c_1 \land c_r, D_1 \land D_r) \).
The extension to a sequence of goals follows the lines of Definition 3.3.1. Let us now prove that this characterization satisfy our requirements.

**Lemma 4.3.12** Goals \( g_1 \) and \( g_2 \) are weakly independent for constraint \( c \), multi-set of constraints \( D \), and program \( P \) in which no atom is allowed to delay iff
\[
\forall \langle c_1, D_1 \rangle \in \text{answer}_P\left(\langle g_1, c, D \rangle \right): \text{there exists a bijection which assigns to each node in a successful branch of } \text{tree}_P\left(\langle g_2, c, D \rangle \right) \text{ a corresponding node in a successful branch of } \text{tree}_P\left(\langle g_2, c, D_1 \rangle \right).
\]

**Proof:** First, assume the bijection exists. Since the nodes are non failure, by Lemma 4.3.5 \( g_1 \) and \( g_2 \) must be weakly independent for \( c, D \), and \( P \). Second, assume that \( g_1 \) and \( g_2 \) are weakly independent for \( c, D \), and \( P \). By Lemma 4.3.9 for all non failure nodes, and in particular those in successful branches of \( \text{tree}_P\left(\langle g_2, c, D_1 \rangle \right) \), there exists a corresponding node in a successful branch of \( \text{tree}_P\left(\langle g_2, c, D \rangle \right) \). Also, by assumption of weak independence, for each node \( r \equiv \langle G_r, c_r, D_r \rangle \) in a successful branch of \( \text{tree}_P\left(\langle g_2, c, D \rangle \right) \) consistent \( (c_1) \) holds, where \( (c_1, D_1) = \text{infer}(c_1 \land c_r, D_1 \land D_r) \).

Let \( \rho \) be a renaming satisfying the conditions in Lemma 4.3.4. By construction, \( \rho((c_1, D_1)) = (c_1, D_1) \) and thus \( \rho((c_0, D_0)) = \text{infer}(c_0 \land \rho(c_r), D_0 \land \rho(D_r)) \). By Lemma 4.3.5, the consistency tests for obtaining the nodes with the same path as \( r \) are performed over a constraint equivalent to \( \rho(c_0) \). Therefore, by Lemma 4.3.8 for each such \( r \) node there is a corresponding node, in a successful branch.

Therefore, we can ensure that:

**Theorem 4.3.13** Let \( g_1 : \cdots : g_n \) be a collection of weakly independent goals for constraint \( c \), multi-set of constraints \( D \), and program \( P \). Let \( g_i, 1 \leq i \leq n \) be a goal such that there exists \( c_i \in \text{answer}_P\left(\langle g_1 : \cdots : g_{i-1}, c, D \rangle \right) \) with \( \text{answer}_P\left(\langle g_i, c_i, D_1 \rangle \right) = \emptyset \). Then, for every \( c_2 \in \text{answer}_P\left(\langle g_1 : \cdots : g_{i-1}, c, D \rangle \right) \) : \( \text{answer}_P\left(\langle g_i, c_2, D_2 \rangle \right) \) = \emptyset.

### 4.3.3 Strong Independence

Strong independence is aimed at detecting goals whose parallelization, when executed in different environments, is guaranteed to be correct and efficient. Thus the definition we are looking for is the following:

**Definition 4.3.14** [strong independence] Goal \( g_2 \) is strongly independent of goal \( g_1 \) for constraint \( c \), multi-set of delayed constraints \( D \), and program \( P \) in which no atoms can be delayed iff
\[
\forall \langle c_1, D_1 \rangle \in \text{answer}_P\left(\langle g_1, c, D \rangle \right) \text{ and } \forall \langle c_r, D_r \rangle \in \text{partial}_P\left(\langle g_2, c \rangle \right) : \text{consistent}(c_0)
\]

where \( c_0 \) is the active constraint obtained by \( \text{infer}(c_1 \land c_r, D_1 \land D_r) \).
The extension to a sequence of goals follows the lines of Definition 3.3.4. Analogously to Theorem 3.3.5, we can show that:

**Theorem 4.3.15** Goal \( g_2 \) is strongly independent of goal \( g_1 \) for constraint \( c \), multiset of delayed constraints \( D \), and program \( P \) in which no atoms can be delayed iff

\[
\forall \langle c_1, D_1 \rangle \in \text{answers}_P(\langle g_1, c, D \rangle):
\]

the search spaces of \( \langle g_2, c, D \rangle \) and \( \langle g_2, c_1, D_1 \rangle \) are the same.

**Proof:** If search space is preserved, by Lemma 4.3.5 \( g_1 \) and \( g_2 \) must be strongly independent for \( c, D, \) and \( P \). On the other hand, assume that \( g_1 \) and \( g_2 \) are strongly independent for \( c, D, \) and \( P \). By Lemma 4.3.9 if search space is not preserved, there must exist a node \( s \equiv \text{fail} \) in \( \text{tree}_P(\langle g_2, c_1, D_1 \rangle) \) obtained by applying \( \rightarrow_{\text{cif}} \) and a node \( r \equiv \langle G_r, c_r, D_r \rangle \) obtained by applying \( \rightarrow_{\text{cif}} \) in \( \text{tree}_P(\langle g_2, c, D \rangle) \) with the same path, such that their parents correspond. Let \( \rho \) be a renaming satisfying the conditions in Lemma 3.2.8. By construction, \( \rho(\langle c_1, D_1 \rangle) \equiv \langle c_1, D_1 \rangle \) and thus, by strong independence, \( \text{consistent}(c_s) \), where \( c_s \) is the active constraint obtained by \( \text{infer}(c_1 \land \rho(c_r), D_1 \land \rho(D_r)) \), must also hold. By Lemma 4.3.4 and Lemma 4.3.5 the consistent test performed for obtaining \( s \) was applied to a constraint equivalent to \( c_s \). This is in contradiction with \( s \) being \( \text{fail} \). ■

As a result of the above theorem, we have that strong independence is not only sufficient but also necessary for ensuring preservation of search space. For reordering, we can also extend Theorem 3.3.7, obtaining:

**Definition 4.3.16** \( \text{[single solution]} \) A goal \( g \) is \text{single solution} for constraint \( c \), multiset of delayed literals \( D \), and program \( P \) iff the state \( \langle g, c, D \rangle \) has at most one finite, non failure derivation in \( P \). ■

**Theorem 4.3.17** If goal \( g_2 \) is both strongly independent of goal \( g_1 \) and \text{single solution} for constraint \( c \), multi-set of delayed constraints \( D \), and program \( P \) in which no atoms can be delayed then \( \forall c_1 \in \text{answers}_P(\langle g_1, c, D \rangle):
\)

\[
\#\text{nodes}_P(\langle g_2 : g_1, c, D \rangle) \leq \#\text{nodes}_P(\langle g_1 : g_2, c, D \rangle).
\]

The proof follows directly from Lemma 4.3.9 and the given operational semantics.

### 4.3.4 Search Independence

As mentioned in Section 3.3.3, in models designed for shared addressing space machines the isolation of environments is not imposed by the machine architecture and thus, in
practice, the goals executing in parallel generally share a single binding environment. Therefore, we need to define a symmetric notion of strong independence, as an extension of that defined in Section 3.3.3:

**Definition 4.3.18** [search independence] Goals \( g_1 \) and \( g_2 \) are search independent for constraint \( c \), multi-set of delayed constraints \( D \), and program \( P \) in which no atom can be delayed iff

\[
\forall \langle c_1, D_1 \rangle \in \text{partial}(\langle g_1, c, D \rangle) \quad \text{and} \quad \forall \langle c_r, D_r \rangle \in \text{partial}(\langle g_2, c, D \rangle) : \text{consistent}(c_s)
\]

where \( c_s \) is the active constraint obtained by \( \text{infer}(c_1 \land c_r, D_1 \land D_r) \).

Then, in the same spirit as Theorem 4.3.15 we can conclude:

**Corollary 4.3.19** Goals \( g_1 \) and \( g_2 \) are search independent for constraint \( c \), multi-set of delayed constraints \( D \), and program \( P \) in which no atom is allowed to delay iff

\[
\forall \langle c_1, D_1 \rangle \in \text{answers}(\langle g_1, c, D \rangle) :
\]

the search spaces of \( \langle g_2, c, D \rangle \) and \( \langle g_2, c_1, D_1 \rangle \) are the same

and \( \forall \langle c_r, D_r \rangle \in \text{answers}(\langle g_2, c, D \rangle) :
\]

the search spaces of \( \langle g_1, c, D \rangle \) and \( \langle g_1, c_r, D_r \rangle \) are the same.

### 4.4 Second Case: \( D \) does not contain atoms

Let us now relax the conditions by allowing atoms to be delayed in the program, but still requiring that the initial multi-set of delayed literals does not contain atoms. We will assume all the conditions imposed on the operational semantics in the previous section, but allowing transitions \( d \rightarrow d \) and \( w \rightarrow w \). Furthermore, the parametric functions \( \text{delay} \) and \( \text{woken} \) should satisfy the following four conditions. The first ensures that there is a congruence between the conditions for delaying an atom and waking it:

\[
(1) \quad a \in \text{woken}(D, c) \text{ iff } a \in D \land \neg \text{delay}(a, c)
\]

The remaining conditions ensure that \( \text{delay} \) behaves reasonably. It should not take variable names into account:

\[
(2) \quad \text{Let } \rho \in \text{Ren}. \quad \text{delay}(a, c) \text{ iff } \text{delay}(\rho(a), \rho(c))
\]

It should only be concerned with the effect of \( c \) on the variables in \( a \):

\[
(3) \quad \text{delay}(a, c) \text{ iff } \text{delay}(a, \exists_{\text{vars}(a)} c)
\]
Also, if an atom is not delayed, adding more constraints should never cause it to delay:

\[
(4) \text{ If } c \to c' \text{ and } delay(a, c), \text{ then } delay(a, c')
\]

Finally, in most practical systems the order in which atoms are delayed and are woken is important. Thus, for conciseness, we will consider that \( D \) is a sequence, rather than a multi-set, and that the order in which literals are added and erased follows the LIFO (last in first out) model.

### 4.4.1 Independence and Search Space Preservation

Let us discuss the and-parallel execution model in this new context. Let \( atoms(D) \) be the subsequence of atoms in \( D \) obtained by eliminating all constraints (note that the relative order among atoms is preserved) and \( cons(D) \) be the constraint formed by the conjunction of all primitive constraints in the sequence of literals \( D \). Assume that given the program \( P \) and the state \( \langle g_1 : g_2 : G, c, D \rangle \) where \( D \) is a sequence of constraints, we want to execute \( g_1 \) and \( g_2 \) in parallel. Then the execution scheme is the following:

- execute \( \langle g_1, c, D \rangle \) and \( \langle g_2, c, D \rangle \) in parallel (in different environments) obtaining the answer constraints \( \langle c_1, D_1 \rangle \) and \( \langle c_r, D_r \rangle \) respectively,
- obtain \( (c_s, D_s) = infer(c_1 \land c_r, cons(D_1) \land cons(D_r)) \)
- execute \( \langle G, c_s, D_r :: D_1 \rangle \).

As a result, the definition of efficiency is the same as that given in the previous section, correctness being extended as follows:

**Definition 4.4.1** Let \( \langle g_1 : g_2 : G, c, D \rangle \) be a state in which \( D \) is a sequence of constraints and \( P \) a program. The parallel execution of \( g_1 \) and \( g_2 \) is correct iff for every \( \langle c_1, D_1 \rangle \in answers_P(\langle g_1, c, D \rangle) \) there exists a renaming \( \rho \in Ren \) and a bijection which assigns to each answer \( \langle c_s, D_s \rangle \in answers_P(\langle g_2, c_1, D_1 \rangle) \) an answer \( \langle c_r, D_r \rangle \in answers_P(\langle g_2, c, D \rangle) \) with \( atoms(D_s) \equiv atoms(\rho(D_r) :: D_1) \) and \( (c_s, cons(D_s)) = infer(c_1 \land \rho(c_r), cons(D_1) \land cons(\rho(D_r))) \).

Note that since we require \( atoms(D_s) \equiv atoms(\rho(D_r) :: D_1) \), and the state \( \langle nil, c_s, D_s \rangle \) is a final state, \( \to_w \) cannot be applied and thus for every atom \( a \) in \( atoms(\rho(D_r) :: D_1) \), \( delay(a, c_s) \) must hold. I.e., every atom left delayed by \( g_1 \) must remain delayed in all finite, non failure derivations of \( \langle g_2, c, D \rangle \) and every atom left delayed by \( g_2 \) must remain delayed in all finite, non failure derivations of
\langle g_2, c_1, D_1 \rangle$. Also note that although the constraints woken during the inference operation have not been erased from the sequence of delayed atoms we still have that \((c_s, \text{cons}(D_1)) = \text{infer}(c_1 \land c_s, \text{cons}(D_1) \land \text{cons}(D_r))\).

As in the previous section, given the definition of search space preservation, it is straightforward to prove the following result:

**Theorem 4.4.2** Let \( P \) be a program, \( \langle g_1 : g_2 : G, c, D \rangle \) be a state in which \( D \) is a sequence of constraints, and \( \langle c_1, D_1 \rangle \in \text{answers}_P(\langle g_1, c \rangle) \). If the search spaces of \( \langle g_2, c, D \rangle \) and \( \langle g_2, c_1, D_1 \rangle \) are the same for \( P \), then

\[
\sum_{i \in TR} K(i) \ast N(i, \langle g_2, c, D \rangle) = \sum_{i \in TR} K(i) \ast N(i, \langle g_2, c_1, D_1 \rangle).
\]

Thus, we can ensure the following:

**Theorem 4.4.3** Let \( P \) be a program and \( \langle g_1 : g_2 : G, c, D \rangle \) be a state in which \( D \) is a sequence of constraints. The parallel execution of \( g_1 \) and \( g_2 \) is efficient iff for every \( \langle c_1, D_1 \rangle \in \text{answers}_P(\langle g_1, c, D \rangle) \) the search spaces of \( \langle g_2, c, D \rangle \) and \( \langle g_2, c_1, D_1 \rangle \) are the same for \( P \).

On the other hand, proving that search space preservation is sufficient for guaranteeing correctness is much more involved. In the previous section, we proved that search space preservation is sufficient for ensuring correctness based on the results of Lemma 4.3.4 and Lemma 4.3.5. However, these lemmas do not apply in the new context. Lemma 4.3.4 guarantees that, in absence of failure, there exists a bijection which assigns to each node \( r \) in \( \text{tree}_P(\langle g_2, c, D \rangle) \) a node \( s \) in \( \text{tree}_P(\langle g_2, c_1, D_1 \rangle) \) with the same path, the sequence of active literals in such nodes being identical up to renaming. This result was used in Lemma 4.3.5 to ensure that, in absence of failure, the constraints selected by the computation rule processed in \( \text{tree}_P(\langle g_2, c, D \rangle) \) and in \( \text{tree}_P(\langle g_2, c_1, D_1 \rangle) \) are the same, up to renaming. Unfortunately, in the new context we cannot guarantee that the same atoms are woken in both executions, and thus the sequence of active literals in nodes with the same path can differ.

**Example 4.4.1** Consider the state \( \langle p(y) : q(x, y), c, \text{nil} \rangle \) and the program \( P \):

\[
\begin{align*}
p(x, y) & \leftarrow y = x. \\
q(x, y) & \leftarrow r(x), s(x, y, w), t(w), y = 0. \\
r(x) & \leftarrow x = 0. \\
s(x, y, w) & \leftarrow w = f(x, z), w = f(0, 1).
\end{align*}
\]
with the following suspension declarations for \( p/2, q/2, r/1 \) and \( s/2 \):

\[
\begin{align*}
? - r(x) & \quad \text{when ground}(x). \\
? - s(x, y) & \quad \text{when ground}(y).
\end{align*}
\]

Figure 4.2 shows (a) the derivation tree from the state \( \{q(x, y), \epsilon, nil\} \) (i.e., \( p(x, y) \) has not been executed) and (b) the derivation tree from the state \( \{q(x, y), y = x, nil\} \) (i.e., \( p(x, y) \) has been executed obtaining in the answer \( \{x = y, nil\} \). It is clear that even though search space is preserved, there is no renaming which makes the sequences of active literals identical. Furthermore, there is also no renaming which makes identical the leftmost literal of every two non failure nodes with the same path. Finally, there is a leftmost literal \( r(x) \) which is not selected to be processed. \( \square \)
The main problem is that, even while search space is still being preserved, relatively different executions can happen. Firstly, an atom \( a \) in \( D_1 \) can be woken during the execution of \( \langle g_2, c_1, D_1 \rangle \) at state \( s_a(a : G, c_a, D_a) \) without changing the search space, if all branches in the tree of \( s_a \) are failure, and the tree starting from the node in \( treeP(g_2, c, D) \) with the same path of \( s_a \), has the same search space. Note that correctness would not be affected since we are talking about failure branches. Secondly, during the execution of \( \langle g_2, c, D \rangle \) and \( \langle g_2, c_1, D_1 \rangle \), the same atoms can be woken in different order, but still preserving the search space and the correctness. Therefore, the same literal can be processed once in \( \langle g_2, c, D \rangle \) and several times in \( \langle g_2, c_1, D_1 \rangle \) or vice versa, and obviously not in nodes with the same path. Note that this can be related to the notion of interleavings in concurrent languages. In some sense, this result is not surprising since dynamically scheduling is similar to concurrency, and therefore interleavings have to be taken into account.

Even with all these problems, we can still prove that search space preservation is sufficient for preserving correctness. The intuition behind this fact is that for finite, non failure branches, every literal processed in \( treeP(g_2, c, D) \) will be sooner or later processed in \( treeP(g_1, c_1, D_1) \) and vice versa. Thus, if search space is preserved, every atom left delayed by \( g_1 \) must remain delayed in all finite, non failure derivations of \( \langle g_2, c, D \rangle \) and every atom left delayed by \( g_2 \) must remain delayed in all finite, non failure derivations of \( \langle g_2, c_1, D_1 \rangle \). On the other hand, it is not longer necessary for ensuring that both correctness and efficiency hold for the parallel execution of the goals, since the failure branches can be both enlarged and pruned involving the same cost without preserving the search space. As a result, it is not longer true either that search space preservation is necessary for ensuring that the number of nodes in the trees are the same. Thus, search space preservation and preservation of the number of nodes cannot be identified any more.

However, proving that search space preservation is sufficient for guaranteeing correctness is long and tedious. We will avoid such exercise\(^2\) due to the existence of another problem which, when solved, will simplify the results. This problem is that although search space preservation guarantees the existence of a bijection between answers, it cannot guarantee that the order in which the sequential answers are obtained will be preserved when the goals are executed in parallel. This is a desirable property when parallelizing a program, since it guarantees that the order intended by the programmer is preserved. In the context of Section 4.3, this preservation comes for free due to the existence of a bijection between answers associated to nodes with the same path. In this new context, the bijection does not necessary apply between such nodes, due to

\(^2\)The proof can be found in [47]
the possible existence of interleadings involving goals which are not single solution.

We can avoid such interleadings by ensuring that for every answer \( \langle c_1, D_1 \rangle \) of \( \langle g_1, c, D \rangle \), no atom in \( D_1 \) is woken during the execution of \( \langle g_2, c_1, D_1 \rangle \), and every atom left delayed (woken) at some point of the execution of \( \langle g_2, c_1, D_1 \rangle \) is also left delayed (woken) at the same point of the execution of \( \langle g_2, c, D \rangle \). Formally, we will say that a node \( s \equiv \langle G_s, c_s, D_s \rangle \) is equivalent w.r.t. delay to a node \( r \equiv \langle G_r, c_r, D_r \rangle \) if for every \( a \in atoms(D_r) : delay(a, c_r) \) iff \( delay(a, c_s) \), and for every \( a \in atoms(D_s \setminus D_r) : delay(a, c_s) \) holds. We will require this condition to be satisfied for every two nodes \( s \) and \( r \) of \( treep(\langle g_2, c_1, D_1 \rangle) \) and \( \rho(treep(\langle g_2, c, D \rangle)) \), respectively, with the same path. Note that this condition is not necessary since it does not allow the interleaving even when one of the goals involved is single solution or it affects branches other than non failure, finite branches. However, if such condition is satisfied, the situation becomes equivalent to the previous one, allowing us to extend all results obtained in the previous section to this new context.

**Lemma 4.4.4** Let \( P \) be a program and \( \langle g_2, c_1, D_1 \rangle, \langle g_2, c, D \rangle \) be two states with an identical sequence of active literals in which \( D \) is a sequence of constraints. There exists a renaming \( \rho \in Ren \) such that for every two non failure nodes \( s \equiv \langle G_s, c_s, D_s \rangle \) and \( r \equiv \langle G_r, c_r, D_r \rangle \) with the same path in \( treep(\langle g_2, c_1, D_1 \rangle) \) and \( \rho(treep(\langle g_2, c, D \rangle)) \), respectively, such that for all their ascendants \( s' \) and \( r' \) of \( s \) and \( r \), respectively, with the same path \( s' \) is equivalent w.r.t. delay to \( r' : G_s \equiv G_r \) and \( atoms(D_r : D_1) \equiv atoms(D_r) \).

**Lemma 4.4.5** Let \( \langle g_2, c_1, D_1 \rangle \) and \( \langle g_2, c, D \rangle \) be two states with an identical sequence of literals such that \( D \) is a sequence of constraints, \( c_1 \rightarrow c \) and \( c_1 \wedge cons(D_1) \rightarrow cons(D) \). Let \( P \) be a program and \( \rho \in Ren \) be a renaming satisfying Lemma 4.4.4. Then, for every two nodes \( s \equiv \langle G_s, c_s, D_s \rangle \) and \( r \equiv \langle G_r, c_r, D_r \rangle \) with the same path in \( treep(\langle g_2, c_1, D_1 \rangle) \) and \( \rho(treep(\langle g_2, c, D \rangle)) \), respectively, such that for all their ascendants \( s' \) and \( r' \), respectively, with the same path \( s' \) is equivalent w.r.t. delay to \( r' : (c_s, D_s) = infer(c_1 \wedge c_r, D_1 \wedge D_r) \).

The proof of the above two lemmas is straightforward given the results of Lemma 4.3.4, Lemma 4.3.5, and the fact that every time that an atom is delayed (woken) in \( r \), it must also be delayed (woken) in \( s \), and vice versa. Therefore, we can ensure the following:

**Theorem 4.4.6** Let \( P \) be a program, \( \langle g_1 : g_2 : G, c, D \rangle \) be a state in which \( D \) is a sequence of constraints, and \( \rho \) be a renaming satisfying Lemma 4.4.4. If for every \( \langle c_1, D_1 \rangle \in answers_P(\langle g_1, c, D \rangle) \), the search spaces of \( \langle g_2, c, D \rangle \) and \( \langle g_2, c_1, D_1 \rangle \) are
the same for \( P \) and for every two non failure nodes \( s \) and \( r \) with the same path in \( \text{treee}_P((g_2,c_1,D)) \) and \( \rho(\text{treee}_P((g_2,c,D))) \), \( s \) is equivalent w.r.t. delay to \( r \), then there exists a bijection which assigns to each final state \( \langle \text{nil}, c_s, D_s \rangle \) in \( \text{treee}_P((g_2,c_1,D_1)) \) a final state \( \langle \text{nil}, c_r, D_r \rangle \) in \( \text{treee}_P((g_2,c,D)) \) with the same path such that \( \text{atoms}(D_s) \equiv \text{atoms}(\rho(D_r) :: D_1) \) and \( (c_s, D_s) = \text{infer}(c_1 \land \rho(c_r), \text{cons}(D_1) \land \text{cons}(\rho(D_r)))) \). ■

4.4.2 Weak Independence

As mentioned before, weak independence aims at characterizing those goals for which the particular form of intelligent backtracking described in Section 3.6.4 can be safely performed. Interestingly, in this new context this characterization does not even need that the preservation of the search space among finite, non failure branches hold. We just need that the conjunction of the answers be consistent. The motivation behind this fact is that, no matter if goals are delayed or woken in different order, if the answers are consistent, it is straightforward to prove that if \( \text{answers}_P((g_i,c_1,D_1)) = \emptyset \) and the sequential execution is able to detect it (i.e., it does not enter in an infinite branch) then \( \text{answers}_P((g_i,c,D)) = \emptyset \). Thus, for every \( (c_2,D_2) \in \text{answers}_P((g_1 : : : g_{i-1},c,D)) \) : \( \text{answers}_P((g_{i+1},c_2,D_2)) = \emptyset \).

As a result, the definition of weakly independent goals is identical to Definition 4.3.11, but allowing the program \( P \) to define literals which can be woken and delayed. Analogously, the results obtained for this kind of goals can be extended to this new context in a straightforward way.

4.4.3 Strong Independence

In this context, strong independence is aimed at detecting goals whose parallelization, when executed in different environments, is guaranteed to be correct, efficient, and preserves the order among answers. Thus the definition we are looking for is the following:

**Definition 4.4.7** [strong independence] Goal \( g_2 \) is strongly independent of goal \( g_1 \) for constraint \( c \), sequence of delayed constraints \( D \), and program \( P \) iff:

\[
\forall \langle c_1, D_1 \rangle \in \text{answer}_P((g_1,c,D)) \forall \langle c_r, D_r \rangle \in \text{partial}_P((g_2,c,D)) :
\]

1. \( \text{consistent}(c_s) \) holds

2. \( \langle \text{nil}, c_s, D_1 \rangle \) is equivalent w.r.t delay to \( \langle \text{nil}, c_r, D_r \rangle \)

where \( c_s \) is the active constraint returned by \( \text{infer}(c_1 \land c_r, \text{cons}(D_1 :: D_r)) \). ■
The definition can be extended to a set of goals analogously to Definition 3.3.4. Thanks to condition (2) and Theorem 4.3.15, it is straightforward to prove not only that strong independence implies search space preservation, but also that the bijection required for correctness holds for nodes with the same path:

**Theorem 4.4.8** If Goal $g_2$ is strongly independent of goal $g_1$ for constraint $c$, multi-set of delayed constraints $D$, and program $P$, then $\forall (c_1, D_1) \in \text{answers}_P([g_1, c, D])$:

- the search spaces of $\langle g_2, c, D \rangle$ and $\langle g_2, c_1, D_1 \rangle$ are the same, and
- there exists a renaming $\rho \in \text{Ren}$ and a bijection which assigns to each final state $\langle \text{nil}, c_s, D_s \rangle$ in $\text{tree}_P([g_2, c_1, D_1])$ a final state $\langle \text{nil}, c_r, D_r \rangle$ in $\text{tree}_P([g_2, c, D])$ with the same path such that $\text{atoms}(D_s) \equiv \text{atoms}(\rho(D_r) :: D_1)$ and $(c_s, D_s) = \text{infer}(c_1 \land \rho(c_r), \text{cons}(D_1) \land \text{cons}(\rho(D_r)))$.

On the other hand, Theorem 4.3.17 does not apply in this new context since, even if goal $g_2$ is both strongly independent of goal $g_1$ and single solution for constraint $c$, multi-set of delayed literals $D$ in which $\text{atoms}(D) \equiv \text{nil}$, and program $P$, there may exist $c_1 \in \text{answers}_P([g_1, c, D])$ for which $\#\text{nodes}_P([g_2 : g_1, c, D]) > \#\text{nodes}_P([g_1 : g_2, c, D])$. For example, if there exists $\langle c_r, D_r \rangle \in \text{answers}_P([g_2, c, D])$, such that an atom in $D_r$ is woken in a failure derivation of $\langle g_1, c_r, D_r \rangle$, and it is not woken during the execution of $\langle g_1, c, D \rangle$, the number of non failure nodes can be greater in $\langle g_2 : g_1, c, D \rangle$.

As a result, we will need the symmetric concept of independence that will be developed in the next section. Given that the number of non failure nodes will be in this case identical after reordering, speed up can only be obtained from this transformation if the amount of work when adding the constraints to the store is in any way reduced.

**4.4.4 Search Independence**

As mentioned before, in models designed for shared addressing space machines the isolation of environments is not imposed by the machine architecture and thus, in practice, the goals executing in parallel generally share a single binding environment. However, in the context of dynamically scheduled languages, it is useful to require that the stack modeling the sequence of delayed atoms remain isolated for each parallel goal. The motivation for this is to allow each parallel agent to easily recognize the atoms left delayed during each parallel execution, and also to simplify the execution in distributed environments.

In this context, we cannot allow an atom in the local stack of a parallel goal to be woken by the constraints added by other parallel execution. Thus, we should extend the definition of search independence as follows:
Definition 4.4.9 [search independence] Goals $g_1$ and $g_2$ are search independent for constraint $c$, multi-set of delayed constraints $D$, and program $P$ in which no atoms can be delayed iff

$$\forall (c_1, D_1) \in \text{partial}_P((g_1, c, D)). \forall (c_r, D_r) \in \text{partial}_P((g_2, c, D)) :$$

1. $\text{consistent}(c_s)$ holds
2. for every $a \in \text{atoms}(D_r)$: $\text{delay}(a, c_r)$ iff $\text{delay}(a, c_s)$
3. for every $a \in \text{atoms}(D_1)$: $\text{delay}(a, c_1)$ iff $\text{delay}(a, c_s)$

where $c_s$ is the active constraint returned by $\text{infer}(c_1 \land c_r, \text{cons}(D_1 :: D_r))$. $\blacksquare$

Note that the last two conditions imply a symmetric notion of equivalence w.r.t. delay. All results obtained in the previous section for search independent goals, can be extended to this new context in a straightforward way.

4.5 General Case

Let us now consider the more general case in which the sequence of delayed literals $D$ might contain atoms. Two problems appear in this new context. The first problem is related to the definition of the and-parallel model and, in particular, to the "conjoin" operation. This operation - conjoining the sequence of delayed atoms associated to the answers obtained in the parallel execution - must be done in such a way that the resulting sequence preserves the order among atoms established by the sequential execution. The existence of atoms in the initial sequence of delayed literals increases the complexity of such operation. Note that the order among delayed constraints is not important. Furthermore, as mentioned before, it is not even necessary to eliminate the woken constraints from the conjoined sequence of delayed literals, although it is convenient for efficiency reasons. The second problem is related to the existence of atoms in $D$ which can be woken by both $g_1$ and $g_2$. This problem can easily be solved by extending the definition of equivalence w.r.t. delay to the following: node $s \equiv \langle G_s, c_s, D_s \rangle$ is equivalent w.r.t. delay to node $r \equiv \langle G_r, c_r, D_r \rangle$ if, for every $a \in \text{atoms}(D_r)$: $\text{delay}(a, c_r)$ iff $\text{delay}(a, c_s)$, for every $a \in \text{atoms}(D_s \setminus D_r)$: $\text{delay}(a, c_s)$ holds, and for every $a \in \text{atoms}(D_r \setminus D_s)$: $\text{delay}(a, c_r)$ holds. I.e., the idea is to require also that all atoms in $D$ not present in $D_1$ remain delayed during the execution of $(g_2, c, D)$. 

88
This solution allows us to solve the first problem by defining the conjoin operation as follows: \( D_r \) is obtained in the and-parallel model as

\[
(D_r \setminus \text{atoms}(D)) :: (D_1 \setminus (\text{atoms}(D \setminus D_r)))
\]

The intuition behind the above operation is that we have to eliminate from \( D_1 \) the atoms woken by \( \langle g_2, c, D \rangle \) (represented by \( \text{atoms}(D \setminus D_r) \)) and then add the atoms left delayed by \( \langle g_2, c, D \rangle \) which do not belong to the initial sequence (represented by \( (D_r \setminus \text{atoms}(D)) \)).

With this solution the results obtained in the previous sections regarding the characteristics of both the search space preservation and the levels of independence can be extended to this new context in a straightforward way. Note that the definition of search space given in the previous section remains the same. This is because the two last conditions of Definition 4.4.9 already imply the extended notion of equivalence w.r.t. delay proposed above.

### 4.6 Ensuring Search Independence “A Priori”

As mentioned in Section 3.4, it is important to determine sufficient conditions which ensure search independence from just the information given by the store rather than from the information given by the partial answers. There are two main reasons for this. First, such sufficient conditions could be tested at run-time just before executing the goals without actually having to execute them (we refer to this as “a priori” detection of independence). Second, the kind of global data-flow analysis required for inferring the information needed to ensure that these sufficient conditions hold may be less complex than that needed for ensuring directly the definition of search independence.

In Section 3.4 we have shown that, in the context of CLP languages, goals \( g_1(\overline{x}) \) and \( g_2(\overline{y}) \) are search independent for a given constraint \( c \) if they are projection independent, i.e., iff

\[
|\overline{x} \cap \overline{y}| \subseteq \text{def}(c) \quad \text{and} \quad (\exists_{\overline{x}} c \land \exists_{\overline{y}} c \rightarrow \exists_{\overline{x} \cup \overline{y}} c)
\]

where \( \text{def}(c) \) denotes the set of variables constrained to a unique value in \( c \). As before, note that \((\exists_{\overline{x}} c \land \exists_{\overline{y}} c \leftrightarrow \exists_{\overline{x} \cup \overline{y}} c)\) is always satisfied.

Intuitively, the condition states that (a) the goals do not have variables in common w.r.t. \( c \) and (b) by projecting the constraint store \( c \) over the variables of each goal we do not lose “interesting” information w.r.t. projecting the original constraint store over the variables of both goals (i.e., the store obtained in the former way entails that obtained in the latter way). This ensures that no matter how \( g_1(\overline{x}) \) and \( g_2(\overline{y}) \) are defined, the
execution of one goal will not be able to modify the domain of the variables of the other, and vice-versa.

Let us now discuss this sufficient condition in the broader context of languages with dynamic scheduling. Consider two goals $g_1$ and $g_2$ and a given constraint $c$ for which the above condition is satisfied. If the sequence of delayed literals $D$ is empty (note that now neither atoms nor constraints are allowed), then we can ensure that the goals are search independent by simply detecting that the above condition holds. The intuition behind this fact is that if there are no delayed literals before the execution of the goals, and they cannot affect the domain of each other’s variables, then their partial answers will be consistent and the instantiation state of their variables will not change no matter if one is executed before or after the other, thus not affecting the literals left delayed by the other goal. Formally:

**Theorem 4.6.1** Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are search independent for a given constraint $c$ and a sequence of delayed literals $D \equiv nil$, if

$$\langle \bar{x} \cap \bar{y} \subseteq \text{def}(c) \rangle \text{ and } (\exists_{\bar{x}} c \land \exists_{\bar{y}} c \rightarrow \exists_{\bar{x} \cup \bar{y}} c)$$

**Proof:** We have to prove that if the condition is satisfied, then for every $\langle c_1, D_1 \rangle \in \text{partialP}(\langle g_1(\bar{x}), c, D \rangle)$ and for every $\langle c_r, D_r \rangle \in \text{partialP}(\langle g_2(\bar{y}), c, D \rangle)$:

1. **consistent**($c_s$) holds, (2) for every $a \in \text{atoms}(D_r)$ delay($a, c_r$) iff delay($a, c_s$), and (3) for every $a \in \text{atoms}(D_1)$ delay($a, c_1$) iff delay($a, c_s$), where $c_s$ is the active constraint returned by infer($c_1 \land c_r, \text{cons}(D_1 :: D_r)$). Clearly, $\text{vars}(D_1) \subseteq \bar{x}$ and $\text{vars}(D_r) \subseteq \bar{y}$. Let $C_1$ and $C_r$ be the constraints added to $c$ during the execution of $\langle g_1(\bar{x}), c, D \rangle$ and $\langle g_2(\bar{y}), c, D \rangle$ yielding $c_1$ and $c_r$, respectively. It is also clear that $\text{vars}(C_1) \subseteq \bar{x}$ and $\text{vars}(C_r) \subseteq \bar{y}$. Thus, the active constraint $c_s$ obtained by infer($c_1 \land c_r, \text{cons}(D_1 :: D_r)$) is in fact $c_1 \land c_r$, and thus $c_s \leftrightarrow c \land C_1 \land C_r$. By Theorem 3.4.3 the condition is satisfied iff the goals are projection independent for $c$. Thus, consistent($c \land C_1 \land C_r$) must hold, and (1) is proved. By assumption of the delay characteristics, delay($a, c \land C_s \land C_r$) iff delay($a, \exists_{\text{vars}(a)}(c \land C_1 \land C_r)$). Since $\text{vars}(C_1) \subseteq \bar{x}$ and $\text{vars}(C_r) \subseteq \bar{y}$, we have that $\exists_{\bar{x} \cup \bar{y}}(c \land C_1 \land C_r) \leftrightarrow \exists_{\bar{x} \cup \bar{y}}(c \land C_s \land C_r)$. Since $\text{vars}(a) \subseteq (\bar{x} \cup \bar{y})$, we have that delay($a, \exists_{\text{vars}(a)}(c \land C_1 \land C_r)$) iff delay($a, \exists_{\text{vars}(a)}(\exists_{\bar{x}} c \land \exists_{\bar{y}} c \land C_1 \land C_r)$) iff delay($a, \exists_{\text{vars}(a)}(\exists_{\bar{x}} c \land \exists_{\bar{y}} c \land C_1 \land C_r)$) iff delay($a, \exists_{\text{vars}(a)}(\exists_{\bar{x}} c \land \exists_{\bar{y}} c \land C_1 \land C_r)$). Since, by assumption, $\exists_{\bar{x}} c \land \exists_{\bar{y}} c \leftrightarrow \exists_{\bar{x} \cup \bar{y}} c$, we have that delay($a, \exists_{\text{vars}(a)}(\exists_{\bar{x}} c \land \exists_{\bar{y}} c \land C_1 \land C_r)$) iff delay($a, \exists_{\text{vars}(a)}(\exists_{\bar{x}} c \land \exists_{\bar{y}} c \land C_1 \land C_r)$). If the atom $a$ is in $\text{atoms}(D_r)$, $\exists_{\text{vars}(a)}(\exists_{\bar{x}} c \land \exists_{\bar{y}} c \land C_1 \land C_r)$ iff $\exists_{\text{vars}(a)}(\exists_{\bar{x}} c \land C_1 \land C_r)$ iff delay($a, c_1$) iff delay($a, c_s$), and (2) is proved. If $a \in \text{atoms}(D_1)$, $\exists_{\text{vars}(a)}(\exists_{\bar{x}} c \land \exists_{\bar{y}} c \land C_1 \land C_r)$ iff delay($a, c_s$) and (3) is proved.  

90
A difference w.r.t. previous cases does arise, however, when the sequence of delayed literals just before the execution of the goals is not empty. In this case the sufficient condition must take into account the constraints established on the variables which appear in the sequence of delayed literals. The reason is that literals woken during the execution of either \( g_1(x) \) or \( g_2(y) \) may introduce new constraints involving variables in both \( x \) and \( y \).

\[
\begin{array}{c}
\text{rit} < q(y), r(x,y) > \\
\text{cit} < y = 2, \xi, r(x,y) > \\
\text{rit} < y = 6, \xi, r(x,y) > \\
\text{fail}
\end{array}
\begin{array}{c}
\text{rit} < q(y), y < 3, \text{nil} > \\
\text{cit} < y = 2, y < 3, \text{nil} > \\
\text{cit} < y = 6, y < 3, \text{nil} > \\
\text{fail}
\end{array}
\]

Figure 4.3:

**Example 4.6.1** Consider the following simple example:

\[ p(x) \leftarrow x = 3. \]

\[ q(y) \leftarrow y = 6. \]

\[ q(y) \leftarrow y = 2. \]

\[ r(x, y) \leftarrow x < y. \]

in which \( r(x, y) \) will be delayed until \( x \) becomes constrained to a unique value.

Figure 4.3 shows (a) the derivations from the state \( \langle q(y), \epsilon, r(x, y) \rangle \) (i.e., \( p(x) \) has not been executed) and (b) the derivations from the state \( \langle q(y), x = 3 \land y < x, \text{nil} \rangle \) (i.e., \( p(x) \) has been executed waking \( r(x, y) \)). It is clear that although \( p(x) \) and \( q(y) \) satisfy the sufficient condition w.r.t. \( \epsilon \) and the empty sequence of delayed atoms, if \( r(x, y) \) is woken before the execution of \( q(y) \) it will prune the search space of \( q(y) \) by making the branch corresponding to \( y = 6 \) fail. □

A similar situation may appear when constraints become active. Thus, new conditions must be developed in order to take the sequence of delayed atoms into account. The solution proposed is to ensure that \( D \) can be partitioned into two sequences in such a way that if we associate those sequences to \( g_1(x) \) and \( g_2(y) \) respectively, the two new goals satisfy the sufficient condition for the given \( \epsilon \) and an empty sequence of delayed literals. While the first sequence corresponds to the delayed goals that depend
on \( g_1(\bar{x}) \), the second one corresponds to those that depend on \( g_2(\bar{y}) \). If there exist delayed literals which depend on neither \( g_1(\bar{x}) \) nor \( g_2(\bar{y}) \), they can be concatenated to any of them.

**Definition 4.6.2** [projection independence] Goals \( g_1(\bar{x}) \) and \( g_2(\bar{y}) \) are projection independent for constraint \( c \) and sequence of delayed literals \( D \) iff \( D \) can be partitioned into two sequences \( D_1 \) and \( D_2 \) such that the goal \( g_1(\bar{x}) : D_1 \) and \( g_2(\bar{y}) : D_2 \) are projection independent for \( c \) and the empty sequence of delayed literals.

**Theorem 4.6.3** Goals \( g_1(\bar{x}) \) and \( g_2(\bar{y}) \) are search independent for constraint \( c \) and sequence of delayed goals \( D \) if they are projection independent for \( c \) and \( D \).

The proof follows directly from Theorem 4.6.1. Note that again, when considering CLP languages without dynamic scheduling, this definition is identical to that defined in [46]. Furthermore, we argue that all sufficient conditions given in [46] are directly applicable to languages with dynamic scheduling by simply transforming the given sequences of delayed literals as proposed above.

### 4.7 Solver Independence

As shown in the previous chapter, in the context of CLP languages, search space preservation is not enough for ensuring the efficiency of any transformation applied to the search independent goals. The reason is that modifying the order in which a sequence of primitive constraints is added to the store may have a critical influence on the time spent by the constraint solver algorithm in obtaining the answer, even if the resulting constraint is consistent. For this reason a new type of independence, *constraint solver independence*, was defined.

This concept is orthogonal to the issue of search space preservation and is only related to the characteristics of the particular constraint solver considered when adding sequences of primitive constraints in different orders. Thus it is tempting to think that the definitions and results obtained in Section 3.5 can be directly applied to languages with dynamic scheduling. However, there is one question which must be considered: the work involved in determining if a goal must become delayed or must be woken.

We believe that a similar approach to that of Section 3.5 can be taken, by considering the solvers which are independent in this sense, those which require the parallel goals, \( g_1 \) and \( g_2 \), to be projection independent for the store \( c \) and the sequence of delayed literals \( D \), and those which also require the goals to be link independent for \( c \) and \( D \). However, this issue certainly needs further study.
4.8 Chapter Conclusions

We have shown how although a simple extrapolation of the CLP-based definitions of independence is correct for CLP languages with passive constraints, it cannot be performed when atoms are dynamically scheduled. The problems posed by the existence of interleavings which preserve the search space, make it more difficult to identify the properties of search space preservation and can affect the order in which the solutions are obtained. Thus, further conditions must be imposed when allowing goals to be run in parallel or reordered. We have proposed a set of conditions and proved them to be sufficient for ensuring both the correctness and efficiency of the parallel model. We have also proposed sufficient conditions for the concepts of independence proposed, which are easier to detect at run-time than the original definitions.
Part II

Global Analysis
Chapter 5

Effectiveness of Global Analysis in Strict Independence-Based Automatic Parallelization

As mentioned in Chapter 2, data-flow analysis is the process of statically – at compile-time – inferring information about the properties of the variables and data structures in a program. The purpose of this process is to use such information to improve the task performed by compilers, program transformation tools, etc. As shown in Section 2.3, abstract interpretation of logic programs allows systematical design of correct data-flow analysis through formalization of the relation between analysis and semantics [38]. The idea is to view data-flow analysis as a non-standard semantics defined over an abstract domain in which the usual domain of values has been replaced by a domain of description of values, and the operations are replaced by corresponding abstract operations defined on the new domain of description.

Much work has been done using the abstract interpretation technique in the context of traditional logic programs (see e.g. [117, 50, 10, 107, 11, 49] and its references). However, only a few studies have been reported which examine the performance of analyzers in the actual optimization task they were designed for (notable exceptions are [155, 105, 133, 146]). In the context of the automatic parallelization of programs within the independent and-parallel model, data-flow analysis seems to be crucial. Unfortunately, little work has been reported on the complete task of global analysis-based compile-time automatic parallelization of logic programs within that model [155, 77]. In this chapter we evaluate the effect of data-flow analysis tools in this context.

We will focus on the automatic parallelization of Prolog programs based on strict independence detection. The choice of Prolog is motivated as follows. Firstly, to our
knowledge, the only implemented parallelizing compiler based on independence and using information provided by global analysis is that found in the &-Prolog system, which parallelizes Prolog programs using as target the &-Prolog language. Secondly, the parallelized programs can actually be executed in this system. This will allow us to compare the results obtained with programs parallelized using different tools. The choice of strict independence is motivated by the fact that many abstract domains have been defined for inferring information useful for detecting strict independence. This is in contrast with parallelization based on non-strict independence, functionality, or other types of safe parallelization conditions. As we will see, groundness and sharing information about the program variables is instrumental in this task. Additionally, freeness and linearity information can also be of great help. Linearity information can significantly improve the accuracy of the sharing information by allowing accurate propagation of definite non sharing among the variables involved in a linear term. The usefulness of freeness in the detection of strict independence comes from its ability to express definite non groundness. Furthermore, it also provides a restricted form of linearity information since definitely free variables are known to be linear.

We will focus our evaluation on five of the already defined abstract domains which infer one or more such properties: the ASub domain defined by Sondergaard [143] for inferring groundness, (pair) sharing and linearity information, the Sharing domain defined by Jacobs and Langen [85] for inferring groundness and (set) sharing information, the Sharing+Freeness domain defined by Muthukumar and Hermenegildo [124] for inferring groundness, (set) sharing and freeness information, and the domains resulting from the combination of the ASub and Sharing domains, and also ASub and Sharing+Freeness domains, as presented by Codish et al [30].

In this chapter we report on the implementation of those abstract domains, their integration in a compiler, and a study of their effectiveness in the automatic parallelization of Prolog programs based on strict independence detection. We construct both local and global analyzers based on the already proposed domains. Then the information gathered by the analyzers is evaluated in terms of its accuracy, i.e., its ability to determine the actual dependencies among the program variables involved in a parallel expression, and its effectiveness, measured in terms of code reduction and also in terms of the ultimate performance of the parallelized programs, i.e., the speedup obtained with respect to the sequential version.

We feel that our presentation and results in this chapter not only provide an empirical assessment of the importance of abstract interpretation techniques in the automatic parallelization task, but also shed new light on the automatic parallelization process itself.
The rest of the chapter is structured as follows. Section 5.1 gives a brief description of the evaluation environment — i.e., the parallelizing compiler. Section 5.2 describes the structure and task of the “annotators” — the actual parallelizers which interface with the analyzers — and the interface itself. Section 5.3 then presents the different domains and the framework they are constructed on, and discusses the usefulness of the information encoded by each domain for the annotation process in terms of the defined interface. Section 5.4 presents the simulator used for performing some of the experiments. Section 5.5 describes the experiments and presents the results obtained from those experiments. Such results are discussed in Section 5.6. Finally, Section 5.7 presents our conclusions.

5.1 The &-Prolog System and Language

As mentioned in the introduction of this chapter, the evaluation has been performed in the context of the &-Prolog system [72]. This system comprises a parallelizing compiler aimed at uncovering goal-level, restricted (i.e., fork and join) independent and-parallelism and an execution model/run-time system aimed at exploiting such parallelism. It is a complete Prolog system, based on the SICStus Prolog implementation, offering full compatibility with the DECsystem-20/Quintus Prolog (“Edinburgh”) stan-
dard. In addition, the \&-Prolog language extensions, the essential part of which will be described shortly, are also supported in the source language. Prolog code is parallelized automatically by the compiler, in a user-transparent way (except for the increase in performance). Compiler switches (implemented as "Prolog flags") determine whether or not code will be parallelized and through which type of analysis. Alternatively, parallel code can be written by the user (the compiler then checking such code for correctness).

The \&-Prolog language is a vehicle for expressing and implementing independent and-parallelism. \&-Prolog is essentially Prolog, with the addition of the parallel conjunction operator \& (used in place of ",", comma- when goals are to be executed concurrently), a set of parallelism-related builtins, which include the groundness and independence tests described in the next section, and a number of synchronization primitives which allow expressing both restricted and non-restricted parallelism. Combining these primitives with the Prolog constructs, such as "+-" (if-then-else), users can conditionally trigger parallel execution of goals. For syntactic convenience an additional construct is also provided: the Conditional Graph Expression (CGE). A CGE has the general form $(i\_cond \Rightarrow goal_1 \& goal_2 \& \ldots \& goal_N)$ where $i\_cond$ is a sufficient condition for running $goal_i$ in parallel under the appropriate notion of independence, in our case strict independence. \&-Prolog if-then-else expressions and CGEs can be nested to create richer execution graphs. One advantage of \&-Prolog is that it subsumes Prolog syntax and semantics and thus it is possible to view the parallelization process as a source to source transformation. Such a transformation is called an annotation.

As shown in Figure 5.1, the \&-Prolog parallelizing compiler used as evaluation environment is composed of several basic modules: global (and local) analyzers inferring information useful for the detection of independence, side-effect and granularity analyzers inferring information which can yield the sequentialization of goals (even when they are independent) based on efficiency or maintenance of observable behavior, annotators which parallelize the Prolog programs using the information provided by the analyzers, etc. The parallelized programs can be executed within the run-time system using one or more processors.

5.2 Annotation Process

As mentioned in Chapter 1, several approaches to automatic parallelization have been proposed. The solution used in the \&-Prolog parallelizing compiler is that proposed initially by R. Warren et al [155] and developed further in [124, 121, 123, 125], which
combines local analysis and run-time checking with a highly sophisticated data-flow analysis based on the technique of abstract interpretation [38].

In this context, the automatic parallelization process is performed as follows. Firstly, if required by the user, the Prolog program is analyzed using one or more global analyzers, aimed at inferring useful information for detecting independence. Secondly, since side-effects cannot be allowed to execute freely in parallel, the original program is analyzed using the global analyzer described in [120] which propagates the side-effect characteristics of builtins determining the scope of side-effects. Finally, the annotators perform a source-to-source transformation of the program in which each clause is annotated with parallel expressions and conditions which encode the notion of independence used. In doing this they use the information provided by the global analyzers mentioned before. Additionally, while annotating each clause, the annotators can also invoke local analyzers in order to infer further information regarding the literals in the clause. In the current implementation, side-effect builtins and procedures are not parallelized. Also, some limited knowledge about the granularity of the goals, in particular the builtins, is used by means of a local analysis. As a result, the only kind of builtins allowed to be run in parallel are the meta-calls.

The annotation process is divided into two subtasks. The first is concerned with identifying the dependencies between each two goals in a clause and generating the minimum number of tests which, when (a priori) evaluated at run-time ensure their independence. Note that, as mentioned in Section 1.3, the detection of independence, based on the information provided by the global analyzers, can be based on non a priori independence conditions. However, if such independence cannot be ensured completely at compile time, then the tests generated for run-time evaluation must be evaluable a priori. The second task is concerned with the core of the annotation process, namely application of a particular strategy to obtain an optimal (under such a strategy) parallel expression among all the possibilities detected in the previous step, hopefully further optimizing the number of tests. In the following we will briefly explain both steps in the particular context of strict independence, in order to better understand the role of the analysis in this process.

5.2.1 Identifying Dependencies

The dependencies between goals can be represented as a dependency graph [36, 83, 94, 99, 22, 123]. Informally, a dependency graph is a directed acyclic graph where each node represents a goal and each edge represents in some way the dependency between the connected goals. A conditional dependency graph (CDG) is one in which the edges are adorned with independence conditions. If those conditions are satisfied, the
dependency does not hold. In an unconditional dependency graph (UDG) dependencies always hold, i.e., conditions are always “false.”

Strict Independence [73] is arguably the most commonly used condition for this purpose. As mentioned in Section 3.2.1, the importance of strict independence lies in the fact that it allows an “a priori” detection of independence. Thus, it can be used even when no information is provided by global analysis. Furthermore, it can be translated into simple tests which can be evaluated efficiently at run-time. For this reason the edges of the CDGs obtained are directly labeled with such tests. In this section we will briefly present how to derive them. These are well understood concepts already present in different ways in the work of [37, 54, 78, 22, 99, 83, 94] and others – here we will mainly follow the presentation of [75].

Recall the notion of strict independence: two goals $g_1$ and $g_2$ are said to be strictly independent for a given substitution $\theta$ iff $\text{vars}(g_1\theta) \cap \text{vars}(g_2\theta) = \emptyset$. A collection of goals is said to be strictly independent for a given $\theta$ iff they are pairwise strictly independent for $\theta$. Also, a collection of goals is said to be strictly independent for a set of substitutions $\Theta$ iff they are strictly independent for every $\theta \in \Theta$. Finally, a collection of goals is said to be simply strictly independent if they are strictly independent for the set of all possible substitutions. This same definition can also be applied to terms, and substitutions, without any change.

**Example 5.2.1:** Let us consider the two goals $p(x)$ and $q(y)$. Given $\theta = \{x/y\}$, we have $p(x)\theta = p(y)$ and $q(y)\theta = q(y)$, so $p(x)$ and $q(y)$ are not strictly independent for this substitution. However, given $\theta = \{x/w, y/v\}$, we have $p(x)\theta = p(w)$ and $q(y)\theta = q(v)$, so $p(x)$ and $q(y)$ are strictly independent for the given $\theta$ because $p(w)$ and $q(v)$ do not share any variable. $\square$

Note that if a term (or a goal) is ground, then it is strictly independent from any other term (or goal). Also, note that strict independence is symmetric, but not transitive.

Given a collection of literals, we would then like to be able to generate at compile-time a condition $\text{cond}$ which, when evaluated at run-time, would guarantee the strict independence of the goals which are instantiations of such literals. Furthermore, we would like that condition to be as efficient as possible, hopefully being more economical than the application of the definition. Consider the set of conditions which includes “true”, “false”, or any set, interpreted as a conjunction, of one or more of the following tests: $\text{ground}(x)$, $\text{indep}(x, y)$ where $x$ and $y$ can be goals, variables, or terms in general. Let $\text{ground}(x)$ be true when $x$ is ground and false otherwise. Let $\text{indep}(x, y)$ be true when $x$ and $y$ do not share variables and false otherwise.
Consider the literals \( g_1, \ldots, g_n \). If no global information is provided, an example of such a correct condition is \( \{ \text{ground}(x) \mid x \in \text{SVG} \} \cup \{ \text{indep}(x, y) \mid (x, y) \in \text{SVI} \} \), where SVG and SVI are defined as follows:

- \( \text{SVG} = \{ v \mid \exists i, j, i \neq j \text{ with } v \in \text{vars}(g_i) \cap \text{vars}(g_j) \} \);
- \( \text{SVI} = \{ (v, w) \mid v, w \notin \text{SVG} \text{ and } \exists i, j, i < j \text{ with } v \in \text{vars}(g_i) \text{ and } w \in \text{vars}(g_j) \} \).

If the above condition is satisfied the literals are strictly independent for every possible substitution, thus ensuring that the goals resulting from the instantiations of such literals will also be strictly independent.

It is easy to see that in general a groundness check is less expensive than an independence check, thus a condition, such as the one given, where some independence checks are replaced by groundness checks is obviously preferable.

Example 5.2.2 : Consider the following literals in a program clause: \( \ldots, a(u), b(x, y), c(z, y), \ldots \) The following table lists all possible series of literals that can be considered for parallel execution, their associated SVG and SVI sets, and a correct local condition with respect to strict independence:

<table>
<thead>
<tr>
<th>Goals</th>
<th>SVG</th>
<th>SVI</th>
<th>i_cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a(u), b(x, y) )</td>
<td>( \emptyset )</td>
<td>( { (u, x), (u, y) } )</td>
<td>( { \text{indep}(u, x), \text{indep}(u, y) } )</td>
</tr>
<tr>
<td>( a(u), c(z, y) )</td>
<td>( \emptyset )</td>
<td>( { (u, y), (u, z) } )</td>
<td>( { \text{indep}(u, y), \text{indep}(u, z) } )</td>
</tr>
<tr>
<td>( b(x, y), c(z, y) )</td>
<td>( { y } )</td>
<td>( { (x, z) } )</td>
<td>( { \text{ground}(y), \text{indep}(x, z) } )</td>
</tr>
<tr>
<td>( a(u), b(x, y), c(z, y) )</td>
<td>( { y } )</td>
<td>( { (u, x), (u, z), (x, z) } )</td>
<td>( { \text{ground}(y), \text{indep}(u, x), \text{indep}(u, z), \text{indep}(x, z) } )</td>
</tr>
</tbody>
</table>

Note that, for efficiency reasons, we can improve the conditions further by grouping pairs in SVI which share a variable \( x \), such as \( (x, y_1), \ldots, (x, y_n) \), by writing only one pair of the form \( (x, [y_1, \ldots, y_n]) \). By following on this idea SVI can be defined in a more compact way as a set of pairs of sets as follows: \( \text{SVI} = \{ (V, W) \mid \exists i, j, i < j, \text{ with } V = \text{vars}(g_i) \setminus \text{SVG} \text{ and } W = \text{vars}(g_j) \setminus \text{SVG} \} \). In many implementations this "compacted" set of pairs is less expensive to check than that generated by the previous definition of SVI. However, in our experiments, and for simplicity, when counting the number of independence checks generated statically we will use the previous definitions of SVI and SVG.

The left-to-right precedence relation for the literals in the clause above can be represented using a directed, acyclic graph in which we associate with each edge which connects a pair of literals the tests for their strict independence, thus resulting in the dependency graph illustrated in Figure 5.2. □

103
The tests generated in the process described above imply the strict independence of the goals for all possible substitutions. However, when considering the literals involved as part of a clause and within a program, the tests can be simplified since strict independence then only needs to be ensured for those substitutions that can appear in that program. This fundamental observation is clearly instrumental when using the results of abstract interpretation-based global analysis in the process of automatic parallelization.

### 5.2.2 Simplifying Dependencies

The annotation process can be improved by using compile-time information about groundness and independence provided either by the user or by an analyzer. This improvement is based on identifying tests which are ensured to either fail or succeed w.r.t. this information: if a test is guaranteed to succeed, it can be reduced to true, thus eliminating the edge; if a test is guaranteed to fail, it can be reduced to false, yielding an unconditional edge.

For any clause $C$, the information known at a program point $i$ in $C$ can be expressed in what we call a *domain of interpretation* $GI$ for groundness and independence: a subset of the first order logical theory, such that each element $\kappa$ of $GI$ defined over the variables in $C$ is a set of formulae (interpreted as their conjunction) containing only predicates of $\text{ground}(x)$ and $\text{indep}(y,z)$, $\{x, y, z\} \subseteq \text{vars}(C)$, and such that $\kappa \not= \text{false}$, and $\forall \kappa \in GI$:

$$
\kappa \supseteq \{\text{ground}(x) \rightarrow \text{indep}(x, y) \mid \{x, y\} \subseteq \text{vars}(C)\} \cup \{\text{ground}(x) \leftrightarrow \text{indep}(x, x) \mid x \in \text{vars}(C)\}
$$

![Diagram](image-url)
For the sake of simplicity, in the rest of the chapter this formula will be assumed to be part of any \( \kappa \), although not explicitly written down. Thus, any time we write \( \kappa = K \), \( \kappa \) should be interpreted as \( K \) augmented with the above set.

For any program point \( i \) of a clause \( C \) where a test \( T_i \) on the groundness and independence of the clause variables is checked, the simplification of such test, based on an element \( \kappa_i \in GI \) over the variables of \( C \), is defined as the refinement of \( T_i \) to yield \( T_i' = \text{improve}(T_i, \kappa_i) \), where:

\[
\text{improve}(T_i, \kappa_i) = \begin{cases} 
  \text{if } \exists t \in T_i \text{ s.t. } \kappa_i \vdash \neg t \text{ then } \text{false} \\
  \text{else if } \kappa_i \vdash T_i \text{ then } \text{true} \\
  \text{else } \{ t \} \cup \text{improve}(T_i \setminus \{ t \}, \kappa_i \cup \{ t \}) 
\end{cases}
\]

Note that there is an implicit restriction on the selection of \( t \in T_i \) in the above definition of \( \text{improve} \): the order in which \( t \) is selected can influence the result of \( \text{improve} \). Consider \( \kappa_i = \{ \text{ground}(x) \rightarrow \text{ground}(y) \} \) and \( T_i = \{ \text{ground}(x), \text{ground}(y) \} \). By selecting first \( \text{ground}(y) \) the final result is \( T_i' = T_i = \{ \text{ground}(x), \text{ground}(y) \} \), whereas by selecting first \( \text{ground}(x) \) the final result is \( T_i' = \{ \text{ground}(x) \} \subset T_i \), which is simpler. We will avoid such non-deterministic behavior by first selecting groundness conditions (because of their lower cost at run-time), then those which does not appear as consequent in any atomic formula of \( \kappa_i \), and then the rest. This will be done following a left-to-right selection rule.

The accuracy and the size (the number of atomic formulae for simple facts) of each \( \kappa \) depend on the kind of program analysis performed. In the next section we will explain how to build this formula from the domains of analysis used in our experiments. Let us now introduce the local analysis.

**Local Analysis**

The simplest kind of information which can be derived comes from a local analysis performed over each clause in isolation. This can be done based on knowledge about the semantics of the builtins and the free nature of the first occurrences of variables. In the case of builtins, their semantics implies certain knowledge about substitutions occurring at the points just before and after their execution. For first occurrences, it can always be ensured that the variable concerned is not ground, up to the point where it first appears.

The information derived by these two sources can be directly expressed in terms of elements of the \( GI \) domain\(^1\). The analysis of clause \( C \) starts with \( Fv_1 \), the set of

\(^1\)Local analysis can also be performed by applying the abstract operations for the domains that will be introduced in the following sections, but only within the scope of the clause. For the sake of simplicity we have preferred to describe it independently of the domains.
variables not occurring in \( \text{head}(C) \), and the formulae for first occurrences of variables, thus:

\[
\kappa_1 = \{ \neg \text{ground}(x) \mid x \in Fv_1 \} \cup \{ \text{indep}(x, y) \mid x \in Fv_1, x \neq y, \text{ and } y \in \text{vars}(C) \}
\]

and proceeds left to right with goals \( g_i \) in the body of \( C \). Assume we have obtained \( \kappa_i \), then \( \kappa_{i+1} \) will be obtained from \( \kappa_i \) and \( g_i \) in the following way:

- \( Fv_{i+1} = Fv_i \setminus \text{vars}(g_i) \)
- if \( g_i \) is not a builtin \( \kappa_{i+1} = \kappa_i \setminus \{ \neg \text{ground}(x) \mid x \in \text{vars}(g_i) \} \cup \{ \text{indep}(x, y) \mid \{x, y\} \cap \text{vars}(g_i) \neq \emptyset \land \{x, y\} \setminus \text{vars}(g_i) \not\subseteq Fv_{i+1} \} \)
- if \( g_i \) is a builtin, let \( \kappa_{bi} \) be the representation for the semantics of \( g_i \) in \( GI \). Then \( \kappa_{i+1} = (\kappa_i \setminus \text{Incons}) \cup \kappa_{bi} \) where \( \text{Incons} \) is the minimum formula s.t. \( \kappa_i \cup \kappa_{bi} \vdash \text{false} \) and \( (\kappa_i \setminus \text{Incons}) \cup \kappa_{bi} \not\vdash \text{false} \)

With such an analysis, the dependencies previously identified can now be simplified, as shown in the example.

**Example 5.2.3** Consider the sequence of literals in Example 5.2.2, augmented with a builtin: \( w \) is \( x+1 \), \( a(w) \), \( b(x, y) \), \( c(z, y) \).

The semantics of \( is/2 \) ensures that both \( x \) and \( w \) are ground after the execution of this builtin. Since this information is downwards closed, the local analysis will be able to derive that this holds not only just after the execution of the builtin, but also at every point in the clause to the right of it. Thus \( \kappa_i = \{ \text{ground}(x), \text{ground}(w) \} \) for all points \( i > 1 \).

The CDG in Example 5.2.2 becomes, by applying the \textit{improve} function with this information, the one shown in Figure 5.3.

```
\[ \text{a}(w) \]
\[ \text{b}(x, y) \rightarrow \text{c}(z, y) \]
```

**Figure 5.3:**

With this simplified CDG the second subtask, i.e., building the parallel expression, is simpler. Furthermore, the \textit{improve} function will also be applied during this second subtask, as we will see.
5.2.3 Building Parallel Expressions

Given a clause, several possible annotations are possible. The second step in the annotation process aims at obtaining an optimal parallel expression among all the possibilities detected in the previous step by applying a particular strategy, hopefully further optimizing the number of tests. Different heuristic algorithms implement different strategies to select among all possible parallel expressions for a given clause.

**Example 5.2.4** Consider again the sequence of literals $a(w)$, $b(x,y)$, $c(z,y)$ in Example 5.2.2. A possible CGE for these goals would be:

$$a(w), (ground(y), indep(x,z) \Rightarrow b(x,y) \& c(z,y))$$

An alternative would be:

$$(indep(w,x), indep(w,z), indep(x,z), ground(y) \Rightarrow a(w) \& b(x,y) \& c(z,y))$$

and yet another alternative:

$$(indep(w,[x,y]) \Rightarrow (a(w) \& b(x,y), c(z,y))$$

; $a(w), (ground(y), indep(x,z) \Rightarrow b(x,y) \& c(z,y))$$

and so on. □

Three different heuristic algorithms are implemented in the &-Prolog system, namely CDG, UDG, and MEL [123]². The CDG algorithm seeks to maximize the amount of parallelism available in a clause, without being concerned with the size of the resultant parallel expression. In doing this, the annotator may switch the positions of independent goals. UDG does essentially the same as CDG except that only unconditional parallelism is exploited, i.e., only goals which can be determined to be independent at compile-time are run in parallel. MEL tries to find points in the body of a clause where it can be split into different parallel expressions (i.e., where edges labeled “false” appear) without changing the order given by the original clause and without building nested parallel expressions. At such points the clause body is broken into two, a CGE is built for the right part of the split sequence, and the process continues with the left part. In the following we will focus on the MEL algorithm, in the particular context of strict independence.

Once an expression has been built, it can be further simplified, unless it is unconditional. Based on the local or global information, the overall condition built by the annotation algorithm can possibly be reduced again.

---

²CDG stands for Conditional Dependency Graph, UDG stands for Unconditional Dependency Graph, and MEL stands for Maximal Expression Length.
Example 5.2.5 Consider once more the clause in Example 5.2.2, augmenting this time the sequence of literals with a different builtin: \( y = f(x,z), \ a(w), \ b(x,y), \ c(z,y). \) Now the analysis can derive \( \kappa_i = \{\text{ground}(x) \land \text{ground}(z) \iff \text{ground}(y)\} \) for all points \( i > 1. \)

Note that, after identifying and simplifying dependencies, the CDG will still be that of Figure 5.2. However, since the groundness of \( y \) implies the groundness of both \( x \) and \( z \) and therefore their independence from any other variable, the annotation of the literals would be:

\[
y = f(x,z), \ (\text{ground}(y) \Rightarrow a(w) \land b(x,y) \land c(z,y))
\]

\( \square \)

5.3 Global Analysis

We have already shown the usefulness of local analysis in the annotation process. Such analyses consider the clauses in isolation instead of as part of a program. Therefore, the information obtained by local analysis is correct for every possible substitution, and the satisfaction of the \( i\text{-conds} \) built using such information ensures the independence of the goals with respect to any substitution. However, for the automatic parallelization the goals are not required to be independent for any substitution, but only for those affecting all the resolvents which contain these goals in any branch of the proof. For such substitutions, simpler \( i\text{-conds} \) may suffice to guarantee the independence of the goals. In order to infer information regarding the substitutions affecting these goals, global data-flow analysis must be performed.

In this section we will first present the abstract interpretation framework used for the evaluation. Then we briefly introduce the definition of each domain and its concretization function, as well as its ability to capture the information needed to simplify the conditions in the parallel expressions.

5.3.1 The Abstract Interpretation Framework PLAI

As mentioned before, abstract interpretation of logic programs allows the systematic design and verification of data-flow analyses by formalizing the relation between analysis and semantics. Therefore, abstract interpretation is inherently semantics sensitive, different semantic definitions styles yielding different approaches to program analysis. For logic programs we distinguish between two main approaches, namely bottom-up analysis and top-down analysis. While the top-down approach propagates the information in the same direction as SLD-resolution does, the bottom-up approach propagates the
information as in the computation of the least fixpoint of the immediate consequences operator $T_P$. In addition, we distinguish between goal dependent and goal independent analyses. A goal dependent analysis provides information about the possible behaviors of a specified (set of) initial goal(s) and a given logic program. In contrast, a goal independent analysis considers only the program itself.

As we have pointed out, in the process of automatic parallelization we are interested in inferring accurate information regarding the substitutions affecting these goals in any proof which can be constructed with the given clause in the given program. It seems that a top-down analysis framework performing goal dependent analysis is the most appropriate for this task. However, it is important to note that recently a number of studies have extended the area of applicability of both the bottom-up and top-down frameworks and their relations with goal dependent and goal independent analysis [116, 59, 27]. In this study we have used a top-down, goal dependent framework, namely the abstract interpretation system PLAI, mainly because of the more mature state of the implementation.

The PLAI abstract interpretation system, is a top-down framework based on the abstract interpretation framework of Bruynooghe [10] with the optimizations described in [125]. Although a detailed description of this system is outside the scope of this chapter, we will point out several features which are relevant to our study, as they either allow efficient analysis or a more effective parallelization.

The framework is based on an abstraction of the (SLD) AND-OR trees of the execution of a program for given entry points. The abstract AND-OR graph makes it possible to provide information at each program point, a feature which is crucial for many applications (such as, for example, reordering, automatic parallelization or garbage collection). For each given goal and abstract call substitution, PLAI builds a node in the abstract AND-OR graph and computes its (possibly many) abstract success substitution(s). Note that, in doing this, PLAI computes the specialized versions (also referred to as multi-variants) for each goal, thus allowing for a quite detailed analysis. The current implementation of the framework allows the user to choose between obtaining a transformed program showing all variants generated for each clause and the particular information inferred for each program point, or the original program in which the information for different variants is collapsed by means of the upper bound operation of the particular abstract domain. Note that, since the framework treats (and takes advantage of) programs in non-normalized form, different call formats of the same predicate yield different goals.

---

3The extension of Bruynooghe's framework and PLAI to the CLP paradigm is the subject of the next chapter.
PLAI implements a highly optimized fixpoint algorithm, defined in [121, 122, 125] (which is essentially equivalent to the algorithm, independently proposed in [98]) for which a distinction between recursive and non-recursive predicates is needed. Briefly, the analysis proceeds as follows. First, a preprocessing of the program is performed to fold disjunctions and determine recursive predicates, then the core of the analysis starts. Non-recursive predicates are analyzed in one pass. For the recursive predicates, non-recursive clauses are analyzed first and once, and the result is taken as a first approximation of the answer. Then, a fixpoint computation for the recursive clauses starts. The number of iterations performed in this computation is reduced by keeping track of the dependencies among nodes and the state of the information being computed. In some cases the fixpoint algorithm is able to finish in a single iteration, thanks to this information.

From the user point of view, it is sufficient to specify the particular abstract domain desired. This information is passed to the fixpoint algorithm, which in turn calls the appropriate abstract functions for the given abstract domain.

5.3.2 ASub Domain

The domain ASub [143] was defined for inferring groundness, sharing, and linearity information. The abstract domain approximates this information by combining two components: definite groundness information is described by means of a set of program variables $D_1 = \mathcal{P}^{\text{Var}}$; possible (pair) sharing information is described by symmetric binary relations on $\mathcal{P}^{\text{Var}} D_2 = (\mathcal{P}^{\text{Var}} \times \mathcal{P}^{\text{Var}})$. The concretization function, $\gamma_{\text{ASub}} : \text{ASub} \rightarrow \mathcal{2}^{\text{Sub}}$, is defined for an abstract substitution $(G, R) \in \text{ASub}$ as follows: $\gamma_{\text{ASub}}(G, R)$ approximates all concrete substitutions $\theta$ such that for every $(x, y) \in \mathcal{P}^{\text{Var}}^2$: $x \in G \Rightarrow \text{ground}(x\theta)$, $x \neq y \Rightarrow \text{indep}(x\theta, y\theta) \neq \emptyset \Rightarrow x \not\rightarrow y$, and $x \not\rightarrow x \Rightarrow \text{linear}(x\theta)$.

Note that the second condition implies that whenever $x \neq y$ if $x \not\rightarrow y$ then we have that $\text{vars}(x\theta) \cap \text{vars}(y\theta) = \emptyset$, and thus $x$ and $y$ are independent.

Let us now present the relation between ASub and the domain GI. Consider an abstract substitution $\lambda_i \in \text{ASub}$ for program point $i$ of a clause $C$. The contents of $\kappa_i$ follow from the following properties of $\lambda_i = (G, R)$ over $\text{vars}(C)$:

- $\text{ground}(x)$ if $x \in G$
- $\text{indep}(x, y)$ if $x \not\rightarrow y$

Note that in this case $\kappa_i$ does not contain either $\neg\text{ground}(x)$ nor $\neg\text{indep}(x, y)$ for every $\{x, y\} \subseteq \text{vars}(C)$, thus no tests in the CDG can ever be reduced to false with only this information.
Example 5.3.1 Consider a clause $C$ such that $vars(C) = \{x, y, z, w, v\}$ and an abstract substitution $\lambda = (\{x\}, \{(z, w), (z, v)\})$. The corresponding $\kappa$ will be: 

\[
\{\text{ground}(x), \text{indep}(y, z), \text{indep}(y, w), \text{indep}(y, v), \text{indep}(w, v)\}.
\]

5.3.3 Sharing Domain

The Sharing domain [84] was proposed for inferring groundness and sharing information. The abstract domain, Sharing = \(2^{\text{PVar}}\), keeps track of set sharing. The concretization function is defined in terms of the occurrences of a variable $U$ in a substitution: \(\text{oocs}(\theta, U) = \{x \in \text{dom}(\theta) \mid U \in vars(x\theta)\}\). If $\text{oocs}(\theta, U) = V$ then $\theta$ maps the variables in $V$ to terms which share the variable $U$. The concretization function $\gamma_{\text{Sharing}} : \text{Sharing} \rightarrow 2^{\text{Sub}}$ is defined as follows:

\[
\gamma_{\text{Sharing}}(\lambda) = \{\theta \in \text{Sub} \mid \forall U \in \text{Var} \cdot \text{oocs}(\theta, U) \subseteq \lambda\}
\]

Intuitively, each set in the abstract substitution containing variables $v_1, \ldots, v_n$ represents the fact that there may be one or more shared variables occurring in the terms to which $v_1, \ldots, v_n$ are bound. If a variable $v$ does not occur in any set, then there is no variable that may occur in the terms to which $v$ is bound and thus those terms are definitely ground. If a variable $v$ appears only in a singleton set, then the terms to which it is bound may contain only variables which do not appear in any other term.

Let us now present the relation between Sharing and the domain GI. Consider an abstract substitution $\lambda_i \in \text{Sharing}$ for program point $i$ of a clause $C$. The contents of $\kappa_i$ follow from the following properties of $\lambda_i$ over $vars(C)$:

- $\text{ground}(x)$ if $\forall S \in \lambda_i : x \not\in S$
- $\text{indep}(x, y)$ if $\forall S \in \lambda_i : x \in S \rightarrow y \not\in S$
- $\text{ground}(x_1) \land \ldots \land \text{ground}(x_n) \rightarrow \text{ground}(y)$ if $\forall S \in \lambda_i : y \in S$ then \(\{x_1, \ldots, x_n\} \cap S \neq \emptyset\)
- $\text{ground}(x_1) \land \ldots \land \text{ground}(x_n) \rightarrow \text{indep}(y, z)$ if $\forall S \in \lambda_i : \{y, z\} \subseteq S$ then \(\{x_1, \ldots, x_n\} \cap S \neq \emptyset\)
- $\text{indep}(x_1, y_1) \land \ldots \land \text{indep}(x_n, y_n) \rightarrow \text{ground}(z)$ if $\forall S \in \lambda_i : z \in S$ then $\exists j \in [1, n], \{x_j, y_j\} \subseteq S$
- $\text{indep}(x_1, y_1) \land \ldots \land \text{indep}(x_n, y_n) \rightarrow \text{indep}(w, z)$ if $\forall S \in \lambda_i : \{w, z\} \subseteq S$ then $\exists j \in [1, n], \{x_j, y_j\} \subseteq S$
The meaning of each implication in $\kappa_i$ can be derived by eliminating the required sets in $\lambda_i$ so that the antecedent of the implication holds, and looking for the new facts $\text{ground}(x)$ or $\text{indep}(x, y)$ in the updated abstract substitution, which now become true. As in $\text{ASub}$, no tests in the CDG can ever be reduced to false with only this information.

Example 5.3.2 Consider the clause $C$ in which $\text{vars}(C) = \{x, y, z, w, v\}$ and the abstract substitution $\lambda = \{(y), (z, w), (z, v)\}$. The corresponding $\kappa$ will be $\{(\text{ground}(x), \text{indep}(y, z), \text{indep}(y, w), \text{indep}(y, v), \text{indep}(w, v), \text{ground}(z) \iff \text{ground}(w), \text{indep}(z, v) \land \text{indep}(z, w) \to \text{ground}(z), \text{indep}(z, v) \to \text{ground}(v), \text{indep}(z, w) \to \text{ground}(w)\}$. Note that $\kappa$ contains all the information derived in Example 5.3.1 plus the information provided by the power of the set sharing for groundness propagation, in contrast with that of the pair sharing representation. □

5.3.4 Sharing+Freeness Domain

The Sharing+Freeness domain [124] aims at inferring $\text{groundness}$, $\text{sharing}$, and $\text{freeness}$ information. The abstract domain approximates this information by combining two components: one $\text{Sh} = 2^{\text{PVar}}$ is the same as the sharing domain; the other $\text{Fr} = 2^{\text{PVar}}$ encodes freeness information. The concretization function $\gamma_{\text{Fr}} : \text{Fr} \to \text{Sub}$ is defined as follows: $\gamma_{\text{Fr}}(\lambda_{fr})$ approximates all concrete substitutions $\theta$ such that for every $x \in \text{PVar}$: if $x \in \lambda_{fr}$ then $\text{free}(x\theta)$.

Let us now present how the information approximated by this abstract domain can be used in improving the CDG obtained in the annotation process. Consider an abstract substitution $\lambda_i \in \text{Sharing+Freeness}$ for program point $i$ of clause $C$. The contents of $\kappa_i$ follow from the following properties of $\lambda_i = (\lambda_{sh}, \lambda_{fr})$ over $\text{vars}(C)$, in addition to those for $\lambda_{sh} \in \text{Sharing}$ which are also applicable from the previous section:

- $\lnot \text{ground}(x)$ if $x \in \lambda_{fr}$
- $\lnot \text{indep}(x, y)$ if $y \in \lambda_{fr}$ and $\forall S \in \lambda_{sh}$ : if $y \in S$ then $x \notin S$
- $\text{ground}(x_1) \land \ldots \land \text{ground}(x_n) \to \lnot \text{indep}(y, z)$ if $z \in \lambda_{fr}$ and $\forall S \in \lambda_{sh}$ : if $\{y, z\} \cap S = \{z\}$ then $\{x_1, \ldots, x_n\} \cap S \neq \emptyset$ and $\exists S \in \lambda_{sh} \{y, z\} \subseteq S$
- $\text{indep}(x_1, y_1) \land \ldots \land \text{indep}(x_n, y_n) \to \lnot \text{indep}(y, z)$ if $z \in \lambda_{fr}$ and $\forall S \in \lambda_{sh}$ : if $\{y, z\} \cap S = \{z\}$ then $\exists j \in [1, n]$, $\{x_j, y_j\} \subseteq S$ and $\exists S \in \lambda_{sh} \{y, z\} \subseteq S$

The intuition behind each implication is as before. The main difference is that now, updating the abstraction $\lambda_i$ for the antecedent to hold, can create an “incoherent” abstraction. In this case $\kappa_i$ allows the simplification of conditions which will always fail. This provides additional precision to that which comes out of the synergistic interaction between the two components of Sharing+Freeness.

112
Example 5.3.3 Consider the same clause $C$ as in Example 5.3.2 and the same sharing component $\lambda_{sh} = \{\{y\}, \{z, w\}, \{z, w\}\}$. Consider the freeness component $\lambda_{fr} = \{w\}$. The corresponding $\kappa$ will be the result of adding the following formulae to the one obtained in the example above: $\{\neg\text{ground}(w), \neg\text{indep}(z, w)\}$. This information, in addition to that derived by $\lambda_{sh}$ makes $\kappa \vdash \neg\text{ground}(z)$. Using this information, any test labeling an edge of a CDG including $\text{ground}(w)$ or $\text{ground}(z)$ or $\text{indep}(z, w)$ can be reduced to false. $\square$

Note that in the example above $\neg\text{ground}(z)$ was derived even though $z \notin \lambda_{fr}$. This is a subtle characteristic of the Sharing+Freeness domain which gives it a significant part of its power. Furthermore, although not directly related to strict independence, the Sharing+Freeness abstract domain is also able to infer definite non freeness for non ground variables: if in the example above $v$ were also an element of $\lambda_{fr}$, then $z$ would be ensured to be bound to a term not only non ground, but also non free. This characteristic is of use in applications such as analysis of programs with dynamic scheduling [106], non-strict independence [19], etc.

5.3.5 Combined Domains

As mentioned before, we have also considered the evaluation the analyzers resulting from the combination of the ASub and Sharing and ASub and Sharing+Freeness domains. This combination, presented by Codish et al in [30], is based on the reduced product approach of Cousot and Cousot [39]. The advantage of this approach is that it allows us to infer more accurate information from the combination without redefining neither the abstract domains nor the basic abstract operations of the original domains. As a result not only is a proof of correctness of the new analyzer unnecessary, but also the gain in accuracy obtained by simply removing redundancies at each step is significant, as shown in [30].

The information approximated by the combined domains can be used to simplify the CDG simply by translating the information inferred by each domain into the $GI$ domain, conjoining the resulting $ss$, and applying the techniques described in previous sections.

5.4 IDRA: Ideal Resource Allocation

We will evaluate the usefulness of the information provided by the analyzers using the speedup obtained w.r.t. the sequential program as the ultimate performance measure. This can be done quite simply by running the parallelized programs in parallel and
measuring the speedup obtained. However, this speedup is limited by the number of processors in the system (and the quality of the scheduler) and thus does not necessarily provide directly useful information regarding the quality of the annotation per se. In order to concentrate on the available parallelism itself, it would be better to be able to determine the speedups for an unbounded number of processors while still taking into account real execution times for the sequential parts and scheduling overheads. A novel evaluation environment, IDRA \cite{57}, has been devised to achieve this.

The &-Prolog system can optionally generate a trace file during an execution. This file is an encoded description of the events that occurred during the execution of a parallelized program. Examples of such events are parallel fork, start goal, finish goal, join, agent busy, etc. Since &-Prolog generates all possible parallel tasks during execution of a parallel program, even if there are only a few (or even one) processor(s) in the system, all possible parallel program graphs, with their exact execution times, can be constructed from this data. IDRA takes as input a real execution trace file of a parallel program run on the &-Prolog system and the time for its sequential execution, and computes the curve of achievable speedup w.r.t. an increasing number of processors.

Note that although this "ideal" parallel execution has been computed, it uses as data a real trace execution file. Real execution times of sequential segments and all delay times are taken into account (including not only the time spent in creating the agents, distributing the work, etc., but also the interruptions of the operating system, etc.), and therefore it is possible to consider the results as a very good approximation to the best possible parallel execution. The approach is similar in spirit to that of AndOrSim \cite{142}, which was shown to produce speedups which closely matched those of the real &-Prolog implementation for the numbers of processors available on the systems in which &-Prolog was run, in that speedups are constructed from data from a real execution. Arguably, the method used in IDRA is potentially more accurate since the unit of measure in AndOrSim was number of resolutions while IDRA uses actual execution time. In fact, the speedups provided by IDRA do correlate well with those of the actual execution.

Table 5.1 illustrates this point by comparing the speedups obtained with the &-Prolog system for a number of programs\footnote{Benchmarks used in the evaluation will be further described in Section 5.5.1.} with those computed by IDRA from traces of the parallelized programs run on a single processor. Logically, IDRA speedups should be an upper bound for the real system and indeed the speedups predicted by IDRA are always larger than those achieved by the actual system. However, while in some cases IDRA estimates are quite close to the observed speedups, a slight divergence is occasionally observed. The latter happens for programs that have very fine granularity,
<table>
<thead>
<tr>
<th>Bench</th>
<th>Query</th>
<th>Speedup</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 2 3 4 5 6</td>
</tr>
<tr>
<td>ann</td>
<td>pa(3) &amp;-Prolog</td>
<td>1.81</td>
<td>2.65 3.22 3.81 4.26</td>
</tr>
<tr>
<td></td>
<td>IDRA</td>
<td>1.80</td>
<td>2.68 3.28 3.95 4.53</td>
</tr>
<tr>
<td>fib</td>
<td>pmain &amp;-Prolog</td>
<td>1.81</td>
<td>2.52 3.13 3.87 4.37</td>
</tr>
<tr>
<td></td>
<td>IDRA</td>
<td>1.99</td>
<td>2.97 3.95 4.91 5.85</td>
</tr>
<tr>
<td>qfib</td>
<td>pmain(10) &amp;-Prolog</td>
<td>1.82</td>
<td>2.70 3.38 4.23 4.65</td>
</tr>
<tr>
<td></td>
<td>IDRA</td>
<td>1.96</td>
<td>2.89 3.81 4.40 5.04</td>
</tr>
<tr>
<td>mmatrix</td>
<td>pm(13) &amp;-Prolog</td>
<td>1.93</td>
<td>2.82 3.80 4.61 5.56</td>
</tr>
<tr>
<td></td>
<td>IDRA</td>
<td>1.98</td>
<td>2.94 3.90 4.80 5.69</td>
</tr>
<tr>
<td>qsortapp</td>
<td>pqs(5) &amp;-Prolog</td>
<td>1.76</td>
<td>2.24 2.55 2.77 2.81</td>
</tr>
<tr>
<td></td>
<td>IDRA</td>
<td>1.76</td>
<td>2.27 2.58 2.81 3.02</td>
</tr>
</tbody>
</table>

Table 5.1: Performance of IDRA

which makes the non-optimal scheduling of the real system have a significant impact. This can be observed by comparing the results for the two versions of “fibonacci”, one of which performs run-time granularity control and thus creates large grain parallelism. The results for this case are much closer to those of IDRA.

5.5 Experimental Results

In this section we present the results of the comparison among the five analyzers (abstract domains and associated abstract functions) which are currently embedded in PLAI: the ASub domain with the abstract functions presented in [26]; the Sharing domain with the abstract functions presented in [125]; the Sharing+Freeness domain and abstract functions defined in [124]; and also the analyzers resulting from the combination of analyzers based on the domains ASub with Sharing and ASub with Sharing+Freeness, as presented by Codish et al in [30]. The aim is to determine the accuracy and effectiveness of the information provided by the analyzers in their application to automatic program parallelization, as well as the efficiency of the analysis process itself.

It could be argued that the experiments are somewhat weak due to the lack of a “best” parallelization with which to establish the comparisons. However, there are many problems when trying to determine what the optimal parallelization might be. In principle, one could think that a parallelization “by hand” performed by the programmer would be the best. However, for complex programs (as many of those used in our experiments), it turns out that the automatic parallelization does better than
what we could do by hand in a reasonable amount of time. Furthermore, it is not easy to prove that a parallelization performed by the programmer is correct. Additionally, such parallelization could not take into account all possible information, as, for example, information regarding the granularity of the goals, since in general, such information depends on the size of the particular arguments given as input. In fact, the best possible parallelization depends on the input and requires using a run-time simulator which runs the program in all possible parallelization schemes, selecting the best from the results obtained in those real executions. However, for most programs this is not practical. Thus we will not attempt to compare parallelizations against an “ideal”, but rather we will perform relative comparisons.

5.5.1 Benchmark Programs

A relatively wide range of programs has been used as benchmarks. Table 5.2 gives good insight into their complexity useful for the interpretation of the results:

- AgV, MV are respectively the average and maximum number of variables in each clause analyzed (dead code is not considered);
- Ps is the total number of predicates analyzed;
- Non, Sim, and Mut are respectively the percentage of predicates non-recursive, simply recursive and mutually recursive;
- Gs is the total number of different goals solved in analyzing the program, i.e., the total number of syntactically different calls.

The number of variables in a clause affects the complexity of the analysis because the abstract functions greatly depend on the number of variables involved. Note that when abstract unification is performed, the variables of both the subgoal and the head of the clause to be unified have to be considered. Therefore, the number of variables involved in an abstract unification can be greater than the maximum number of variables shown in the table. The number of recursive predicates affects the complexity of the fixpoint algorithm possibly increasing the number of iterations needed.

5.5.2 Efficiency Results

Table 5.3 presents the efficiency results in terms of analysis times in seconds (Sparc-Station 10, one processor, SICStus 2.1, native code). It shows for each benchmark and analyzer the average times out of ten executions. In the following S (set sharing)
<table>
<thead>
<tr>
<th>Bench.</th>
<th>AgV</th>
<th>MV</th>
<th>Ps</th>
<th>Non</th>
<th>Sm</th>
<th>Mut</th>
<th>Gs</th>
</tr>
</thead>
<tbody>
<tr>
<td>aiakl</td>
<td>4.58</td>
<td>9</td>
<td>7</td>
<td>42</td>
<td>57</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>ann</td>
<td>3.17</td>
<td>14</td>
<td>65</td>
<td>43</td>
<td>20</td>
<td>36</td>
<td>73</td>
</tr>
<tr>
<td>bid</td>
<td>2.20</td>
<td>7</td>
<td>19</td>
<td>68</td>
<td>31</td>
<td>0</td>
<td>27</td>
</tr>
<tr>
<td>boyer</td>
<td>2.36</td>
<td>7</td>
<td>26</td>
<td>73</td>
<td>3</td>
<td>23</td>
<td>29</td>
</tr>
<tr>
<td>browse</td>
<td>2.63</td>
<td>5</td>
<td>8</td>
<td>12</td>
<td>62</td>
<td>25</td>
<td>9</td>
</tr>
<tr>
<td>deriv</td>
<td>3.70</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>fib</td>
<td>2.00</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>grammar</td>
<td>2.13</td>
<td>6</td>
<td>6</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>hanoiapp</td>
<td>4.25</td>
<td>9</td>
<td>2</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>mmatrix</td>
<td>3.17</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>occur</td>
<td>3.12</td>
<td>6</td>
<td>4</td>
<td>25</td>
<td>75</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>peephole</td>
<td>3.15</td>
<td>7</td>
<td>26</td>
<td>46</td>
<td>7</td>
<td>46</td>
<td>28</td>
</tr>
<tr>
<td>progeom</td>
<td>3.59</td>
<td>9</td>
<td>9</td>
<td>33</td>
<td>66</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>qplan</td>
<td>3.18</td>
<td>16</td>
<td>46</td>
<td>39</td>
<td>32</td>
<td>28</td>
<td>51</td>
</tr>
<tr>
<td>qsortapp</td>
<td>3.29</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>query</td>
<td>0.19</td>
<td>6</td>
<td>4</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>rdtok</td>
<td>3.07</td>
<td>7</td>
<td>22</td>
<td>31</td>
<td>27</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>read</td>
<td>4.20</td>
<td>13</td>
<td>24</td>
<td>54</td>
<td>12</td>
<td>33</td>
<td>47</td>
</tr>
<tr>
<td>serialize</td>
<td>4.18</td>
<td>7</td>
<td>5</td>
<td>20</td>
<td>80</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>tak</td>
<td>7.00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>warplan</td>
<td>2.47</td>
<td>7</td>
<td>29</td>
<td>51</td>
<td>31</td>
<td>17</td>
<td>36</td>
</tr>
<tr>
<td>witt</td>
<td>4.57</td>
<td>18</td>
<td>77</td>
<td>42</td>
<td>35</td>
<td>22</td>
<td>96</td>
</tr>
<tr>
<td>zebra</td>
<td>2.06</td>
<td>25</td>
<td>6</td>
<td>66</td>
<td>33</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 5.2: Benchmark Profiles

denotes the analyzer based on the Sharing domain, P (pair sharing) denotes the analyzer based on the ASub domain, SF (set sharing + freeness) denotes the analyzer based on the Sharing domain, and P×S and P×SF denote the analyzers based on the combination of P and S, and P and SF, respectively.
Table 5.3: Analysis Times

Table 5.4 presents efficiency results in terms of memory consumption. For each benchmark and analyzer it shows the number of Kbytes for the global stack segment created in the process (showing the size of terms created) and the heap created in the process (showing the size of data asserted). All measurements have been made disallowing garbage collection during the analysis. Corresponding results for the local stack, choice-point stack and trail have been obtained but are not included. The reason is that for the local stack the amount of memory used is negligible compared to those of global stack and heap (never more than 69 Kbytes, and usually less than 10). Also, the number of choice-points created and backtrakings performed is almost zero.
One way to measure the accuracy and effectiveness of the information provided by abstract interpretation-based analyzers is to count the number of CGEs which actually result in parallelism, the number of these which are unconditional, and the number of groundness and independence tests in the remaining CGEs, which provides an idea of the overhead introduced in the program. The results are shown in Table 5.5 and Table 5.6. Benchmarks have been parallelized using the MEL annotator in the following different situations: without any kind of information (“N” in the table), with information
from the local analysis ("L"), and with that provided by each of the global analyzers. The results for the combined analyzers are in all but five cases as good as those for the best of the analyzers combined. The exceptions are shown in a separate table. Note that to obtain the results we inhibited local analysis so as to measure the power of the global analyzers by themselves.

<table>
<thead>
<tr>
<th>Bench. Program</th>
<th>Total CGEs</th>
<th>Uncond. CGEs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N  L  S  P</td>
<td>N  L  S  P</td>
</tr>
<tr>
<td>aiakl</td>
<td>2  2  2  2</td>
<td>0  0  0  0</td>
</tr>
<tr>
<td>ann</td>
<td>28 14 26 26</td>
<td>12 0 0 0 0</td>
</tr>
<tr>
<td>bid</td>
<td>8  6  8  8</td>
<td>5  0  3  5</td>
</tr>
<tr>
<td>boyer</td>
<td>3  2  3  3</td>
<td>2  0  0  0</td>
</tr>
<tr>
<td>browse</td>
<td>9  5  5  5</td>
<td>4  0  0  0</td>
</tr>
<tr>
<td>deriv</td>
<td>5  4  4  4</td>
<td>0  0  0  4</td>
</tr>
<tr>
<td>fib</td>
<td>1  1  1  1</td>
<td>0  1  1  1</td>
</tr>
<tr>
<td>grammar</td>
<td>4  0  3  3</td>
<td>0  0  0  0</td>
</tr>
<tr>
<td>hanoiapp</td>
<td>1  1  1  1</td>
<td>0  0  0  0</td>
</tr>
<tr>
<td>mmatrix</td>
<td>2  2  2  2</td>
<td>0  0  0  2</td>
</tr>
<tr>
<td>occur</td>
<td>3  3  2  2</td>
<td>0  1  1  2</td>
</tr>
<tr>
<td>peephole</td>
<td>11 2 11 11</td>
<td>0  0  1  1</td>
</tr>
<tr>
<td>progeom</td>
<td>3  2  3  3</td>
<td>0  0  0  0</td>
</tr>
<tr>
<td>qplan</td>
<td>31 20 31 31</td>
<td>18 0 3  3</td>
</tr>
<tr>
<td>qsortapp</td>
<td>1  1  1  1</td>
<td>0  0  0  1</td>
</tr>
<tr>
<td>query</td>
<td>2  2  2  2</td>
<td>0  0  0  1</td>
</tr>
<tr>
<td>rtdtok</td>
<td>0  0  0  0</td>
<td>0  0  0  0</td>
</tr>
<tr>
<td>read</td>
<td>2  1  2  2</td>
<td>0  0  1  1</td>
</tr>
<tr>
<td>serialize</td>
<td>2  1  2  1</td>
<td>0  0  0  0</td>
</tr>
<tr>
<td>tak</td>
<td>1  1  1  1</td>
<td>0  1  0  0</td>
</tr>
<tr>
<td>warplan</td>
<td>16 11 14 14</td>
<td>9  0 1  0</td>
</tr>
<tr>
<td>witt</td>
<td>39 24 39 39</td>
<td>24 0 2  5</td>
</tr>
<tr>
<td>zebra</td>
<td>4  3  3  3</td>
<td>2  0 1  1</td>
</tr>
</tbody>
</table>

Table 5.5: Results for Effectiveness — Static Tests

The results for each benchmark and each of the situations are shown in tables 5.5, 5.6, and 5.7. The tables give the total number of CGEs obtained, the number which are unconditional, and the total number of groundness and independence tests for those which are not unconditional.

120
The results for a representative subset of the benchmarks used are presented in figures 5.4 and 5.5. For each benchmark, a diagram with speedup curves obtained with
IDRA is shown. Each curve represents the speedup achievable for the parallelized version of the program obtained with the MEL annotator in one of the situations shown in the static tests. A curve has been labeled with more than one situation when either the resulting parallelized programs where identical or the differences among the speedups obtained were negligible (i.e., impossible to distinguish by looking at the diagram).

5.6 Discussion

The efficiency results in terms of time required by the analysis (Table 5.3) suggest that the analysis process is reasonably efficient (all the analysis code is written in Prolog). Typically, the analysis takes less than 2 or 3 seconds. The longest execution (Sharing for ann) takes 19.36 seconds which is still not unreasonable, considering the complexity of the benchmark.

The results from looking at Table 5.3 appear inconclusive at first sight. The reason is the high number of parameters involved which, for simplicity, are not included in the table: number of specializations, number of recursive and mutually recursive predicates, number of iterations in each computation, number of variables involved in each abstract unification, etc.

Some conclusions can still be inferred from Table 5.3 in some cases. For example, the above mentioned parameters have similar values in a number of cases represented by bid, deriv, fib, hanoiapp, mmatrix, occur, qsortapp, and tak. In these cases the relative complexity of the analyzers is then clearly reflected in the figures in the table: the abstract operations of the Sharing+Freeness analysis are more complex than those of Sharing (since it has an additional component) and these in turn are much more complex than those of ASub.

However, in general the tradeoffs are much more complex than implied by the
The important intervening factor is accuracy. An accurate analysis generally computes smaller abstract substitutions thus reducing the time needed for the abstract operations. Accuracy also greatly affects the fixpoint computation: its absence results in more iterations, specializations, etc. This effect can be observed in a number of cases in which the lack of groundness propagation in the ASub analyzer greatly affects efficiency: aiakl, qplan and witt. In these benchmarks, the total number of iterations within fixpoint computations for ASub is approximately 6.5 times that of the other analyzers, and in the last two benchmarks the number of specializations increases by 2.5 times. Conversely, there are other cases (e.g., ann, boyer, serialize and warplan) in which the Sharing or the Sharing+Freeness analyzers take much longer than ASub due to the lack of (accurate) linearity information.\footnote{As we will see in the next chapter, in all the cases these shortcomings are alleviated in the corre-}

Figure 5.4: Results for Effectiveness — Dynamic Tests I
Comparing the time spent and the memory consumed in the analysis process it is clear that one accurately follows the other, i.e., high memory consumption often indicates a long execution time and vice versa. This is especially true when considering the global stack, where most of the memory consumption takes place. Memory consumed in the heap (where the memo table is stored) is almost negligible compared to that of the global stack, and it is heavily related to the number of specializations which occur in each analysis. The fact that the analyzers do not consume much heap space has been a big surprise considering the heavy use of the database performed during the analysis. Since global stack consumption is quite related to the size of the substitutions each analyzer handles it can be concluded that the size of the (representations of the) abstract substitutions dominates the consumption of memory (and time) by the analyzers. Nonetheless, the more specializations and fixpoint iterations, the more substitutions the analyzer has to handle, and this, as already mentioned, depends on the accuracy inherent in each domain.

Regarding the effectiveness of the information inferred by each analyzer, there are two key issues to be studied: whether the results of the analysis are effective in eliminating CGEs which have a test that will always fail, and whether they are effective in eliminating tests that will always succeed. With respect to the first point, tables 5.5 and 5.6 show that definite non-groundness and definite sharing, achieved in the case of the Sharing+Freeness analysis due to the combination of sharing and freeness, is quite effective. While Sharing and ASub can only help in eliminating CGEs by identifying corresponding combined domains, so that the time is less than the expected sum of the times of each original component.
dead code (which is not parallelized) the local analysis (unable to detect dead code) is able to eliminate more CGEs than either of them in a fair number of cases: ann, bid, boyer, peephole, qplan, warplan, witt. Sharing+Freeness proves to be the most accurate at this task, giving always the least number of CGEs. It is important to note that although some elimination of CGEs was expected at the beginning of the study, the actual impact of the results of this type of analysis is quite surprising: the Sharing+Freeness analysis can reduce the number of CGEs in 16 out of 23 benchmarks (the complete set used), and the reduction is often of half or more of the CGEs created without analysis.

However, it is when considering the simplification of the conditions in the CGEs that global analysis shows its power: even in the cases where Sharing or ASub have to deal with more CGEs than the local analysis, the total number of tests is usually less. Regarding the comparison among the different global analyzers, it is clear that ASub is better than Sharing (at least for independence checks, though not for groundness!) and that Sharing+Freeness is the best by far in terms of accuracy, although often at a cost in analysis time. This can be surprising when noticing that the sharing information provided by ASub is usually more accurate than that of Sharing+Freeness as shown in [30]. This apparent contradiction is solved when considering the amount of information provided by the set sharing information, already pointed out when translating this information into the GI domain, which allows the annotators to significantly simplify the tests for parallelization. The combined analyzers always obtain the same number of tests as those of the best of the analyzers combined, and, in a few cases, slightly better results are obtained. The number of tests obtained by ASub is reduced when combined with Sharing in three cases: ann, qplan, and witt. This is not surprising since the last two are in the class of programs for which ASub loses information. In the case of ann, the advantage is due to the ability of the Sharing domain to infer independence of two variables from the independence of others, an ability which ASub lacks. There are two exceptions for Sharing+Freeness combined with ASub: serialize, and boyer. In both of them an independence check is eliminated, thanks to the more accurate linearity information provided by ASub.

An important conclusion from the study is the importance in the application of non-groundness information (represented by freeness in our case) in addition to that of sharing and groundness. The Sharing+Freeness domain turns out to be quite sufficient in this sense, offering acceptable results in most cases. However, in some cases the results from Sharing+Freeness can be improved coupling it with the ASub domain, a combination which gives the absolute best results for the domains considered. ASub and Sharing gave reasonable and similar overall results, with a relatively large advantage.
for one or the other in some cases.

Finally, we discuss the effectiveness of the analysis process in terms of the ultimate objective of the application, i.e., in terms of actual speedups obtained from the parallelization of the program. Note that the results include the overhead of running the independence checks and thus slow-downs can be observed. Slow-downs can always be avoided by using the UDG annotator, which only allows unconditional CGEs. However, only the results for MEL are presented here, since they are more interesting from the point of view of observing the performance of the analysis.

The first observation is that speedups obtained for a given benchmark do reflect in some ways the accuracy results. Accordingly, the overall results favor the Sharing+Freeness analysis. However, there are exceptions to this. In particular, in the case of ann, better results can be observed for all other analyzers except local. The reason for this is interesting: it is due to a particular clause being annotated in two different ways. With most of the analyzers, a CGE with a groundness test is built. The better information obtained by the Sharing+Freeness analysis allows eliminating this groundness test because it will always fail, and a new CGE with a number of independence checks is then built. It turns out that all tests will ultimately fail at run-time. However, with Sharing+Freeness the independence test, which turns out to be much more complex, is performed. The other analyzers, being less accurate, do not eliminate the groundness test, which turns out to fail early and thus gives better performance. Both ASub and the local analysis perform as well as Sharing+Freeness in some cases. Sharing also behaves sometimes as well as Sharing+Freeness, but in those cases ASub also does. Thus, ASub also proves to be quite powerful.

Although studying the tradeoffs among the different annotators is beyond the scope of this study, we stress that UDG avoids slow-down situations, which makes it an obvious choice for completely transparent parallelization. The drawback is that sometimes no speedup is obtained (for example, for ann). The differences between MEL and CDG are interesting as the curves for hanoiapp (parallelized using MEL) and hanoiapp-cdg (parallelized using CDG) illustrate. MEL correctly but inefficiently parallelizes a call to hanoi and a call to append, while CDG parallelizes a call to hanoi with a sequence composed of the other call to hanoi and a call to append. For a comparison of the different annotators see [17, 15].

A final issue is the importance that a few checks (out of the large collection that may appear in a program) may have. This was already illustrated in the case of ann but is also relevant in other cases. In deriv, the important differences in speedups are due to only four independence checks. In occur a significant difference can be observed between Sharing+Freeness and no analysis that is only due to two ground-
ness and four independence checks. And in mmatrix the significant difference between Sharing+Freeness and ASub is due to only two independence checks. On the other hand in aiakl and bid no significant difference in speedup is observed despite variations of ten independence and five groundness checks respectively.

In summary, the experiments confirm the importance of global data-flow analysis in the parallelization task. Inevitably, also small speedups (or even slow-downs, when not using UDG) are obtained for some benchmarks, such as aiakl, bid and boyer due to their lack of parallelism based on strict independence. We hope that using more general independence conditions, such as the conditions given in previous chapters, will allow us to extract parallelism in even more cases.

5.7 Chapter Conclusions

We have studied the effectiveness of global analysis in the parallelization of logic programs using strict independence by selecting a number of well-known approximation domains, building analyses based on them, embedding such analyses in a complete parallelizing compiler, and studying the performance of the resulting system. Although speedups can be obtained in some cases using only local analysis our overall conclusion, based on the improvements observed, is that global analysis based on abstract interpretation is indeed a very powerful tool in this application, and thus our objective of adapting this technique to the realm of constraint logic programs with dynamic scheduling appears well justified.
Chapter 6

Analyzing Constraint Logic Programs

As mentioned in the introduction, the constraint logic programming (CLP) paradigm [86] is a relatively recent proposal which has emerged as the natural combination of the constraint solving and logic programming paradigms. Thus, constraint logic programming significantly extends the expressive power of logic programming to, for example, finite domain and numerical problems, previously the turf of imperative languages. In this context, traditional logic programming (LP) can be seen as an instance of CLP in which constraints are equations over terms and constraint solving is done by the well known unification.

Often, the efficiency of CLP programs is better than that obtained by an equivalent program written in another language, specially when the program can take advantage of constraint-based pruning of the search space. However, sometimes the expressive power is paid in terms of efficiency. Such performance limitations, combined with the increasing acceptance of these languages, have motivated a growing interest in data-flow analysis based optimization techniques for CLP languages, and, in particular, in the application of abstract interpretation [38].

Much work has been done using the abstract interpretation technique in the context of LP (e.g., [117, 50, 10, 107, 49]). A number of practical systems have been built, some of which have shown great usefulness and practicality [133, 155, 125, 49, 23]. Thus, it is natural to expect that this technique should also be useful in the context of CLP. A few general frameworks have already been defined for this purpose [108, 32, 12, 60]. However, one common characteristic of these frameworks is that they are either not implementation oriented or they depart from the approaches that have been so far quite successful in the analysis of traditional logic programming (LP) languages. It is the
point of this chapter to show how some of the techniques which have been used to great success and practicality in LP, and for which efficient fixpoint algorithms have already been developed, can relatively easily be extended to the analysis of CLP programs.

The point above is illustrated by proposing a simple but quite general and powerful extension of Bruynooghe’s traditional framework in order to make it applicable to the analysis of CLP programs. We also extend the framework to deal with non-normalized programs and with passive constraints. Finally, we give correctness conditions for the resulting framework. The resulting description represents a fully specified algorithm for analysis of CLP programs.

Then, as an example of the use of this approach, a complete, constraint system independent abstract analysis is presented for approximating definiteness information. In doing this it keeps track of definite dependencies among variables in order to accurately propagate definiteness information. The analysis is of quite general applicability since it uses in its implementation only constraints over the Herbrand domain. The abstract domain has been implemented within the abstract interpretation system PLAI. This system is based on the framework of Bruynooghe [10], optimized with the specialized domain-independent fixpoint defined in [121, 122, 125], and generalized to support analysis of practical CLP languages, following the guidelines presented in this chapter. Results from this implementation showing its efficiency and accuracy are also presented.

The rest of the chapter proceeds as follows: section 6.1 discusses other approaches to the analysis of CLP languages. Section 6.2 describes the modified operational semantics of CLP programs considered. Section 6.3 presents the extension of Bruynooghe’s framework to the analysis of CLP programs. Section 6.4 then presents the application of the framework to the approximation of definiteness information. Some results from the implementation of this analysis are then presented and discussed in Section 6.5. Finally, section 6.6 presents our conclusions.

6.1 Towards a CLP Analysis Framework

There has been considerable interest in developing new abstract interpretation frameworks for CLP languages. To our knowledge, at least four frameworks have been proposed previously or simultaneously with our work.\footnote{This idea illustrated in this chapter were first presented at the ICLP'91 Workshop on Constraint Logic Programming.} Marriott and Søndergaard [108] present a general and elegant, semantics based framework. It is based on a definition-independent meta-language which can express the semantics of a wide variety of programming languages, including CLP languages. However, from a practical
point of view, this framework does not provide much simplification to the developer of the abstract interpretation system, in the sense that many issues are left open.

In fact, one of the advantages of the most popular methods used in the analysis of conventional LP systems (for example Bruynooghe’s method [10] and the optimizations proposed for it [125]) is that they are both fully specified and “generic,” in the sense that they specify much of what is needed leaving only the definition of the domain, domain dependent functions, and assurance of correctness criteria to be provided by the implementor. It is our intention to develop a framework for CLP program analysis at this level of specification.

Codognet and Filé [32] also present a quite general framework for the description of both CLP languages and their static analyses and an implementation approach. Although more concrete, this proposal is still more abstract than the level pointed out above as our objective. On the other hand this paper introduces the quite interesting idea of implementing the abstract functions actually using constraint solvers, to which we will return later.

Giacobazzi et al [60] formulate a general algebraic framework for constraint logic programming. It formulates the operational and fixpoint semantics within this framework and shows that abstract interpretation is simply an other instance of the general framework which safely approximates the instance given by the concrete constraint system. Also, this work is presented in fairly general terms and does not offer an implementation approach to the application developer.

Finally, Bruynooghe and Janssens [12] present a specialized framework (which was developed in parallel with the proposal presented in this chapter) which is based on the idea of adding complexity to the framework with the potential benefit of decreased complexity in the abstract domain. This is done by incorporating a local form of “suspension” so that some goals can be reconsidered if later execution in a different environment can provide further information. This extension is based on a particular view of the execution of a CLP program in which constraints are considered as goals which can suspend depending on the state of their arguments and on the particular constraint system.

The view of constraints as suspended goals could be interesting and worth pursuing. However, this makes it more difficult to make the framework fully general and we prefer to take the more traditional notion presented in the CLP scheme (as introduced in the previous sections) in which constraints take the place of substitutions and goals always either succeed or fail, in the former case possibly placing new constraints.\(^2\)

\(^2\)In fact, actual suspension, as is often used in the solving of non-linear arithmetic constraints or in programs with explicit coroutining can also be modeled in this way. However, we propose treating
One of the main points of this chapter is to show that standard abstract interpretation frameworks for logic programs are essentially still useful for the analysis of constraint logic programs, provided the parts that relate to the abstraction of the Herbrand domain and unification functions are suitably generalized. Indeed, in this traditional view the role of goals and their control are basically identical to those in traditional LP systems, the differences being essentially limited to replacing the notions of Herbrand domain, unification, and substitutions by those of constraint system, conjunction, and constraints.

In particular, we argue that the traditional framework of Bruynooghe and its extensions can be used for analyzing constraint logic programs by using the notions of abstract constraint and abstract conjunction and reformulating the safety conditions, but keeping the construction of the AND-OR tree, the implementation and optimizations of the fixpoint algorithm, the notions of projection and extension, etc. This has the advantage that the relatively large number of implementations based on this scheme or derivations thereof can be applied to CLP systems provided the safety conditions and other related requirements proposed herein are observed.

6.2 Modifying the CLP Operational Semantics

We assume the requirements of operational semantics of the CLP languages presented in Section 3.1 but, for practical reasons, we allow the literals to be in non normalized form. The assumption regarding passive constraints is not really necessary. It has been required in order to maximize the similarity with the original analysis framework for logic programs of Bruynooghe [10]. The relaxation of this assumption will be discussed in Section 6.3.4 and subsumed by the framework presented in the next chapter.

We now modify this operational semantics in order to obtain an equivalent operational semantics in which states are of the form \( s = \langle G, c \rangle \) in which \( G \) is either the tail of a body of a clause or the tail of the initial query, and \( c \) is defined over the variables of \( G \). These two modifications, proposed in [10], provide the basis for the usefulness and practical success of the original analysis framework. Firstly, if \( G \) is either the tail of a body of a clause or the tail of the initial query, the states can be organized in an AND-OR tree. These concrete AND-OR trees can be abstracted as abstract AND-OR graphs. Each graph represents a set of proof trees originating from a set of calls to a top-level predicate and gives rise to multiple specializations in a very natural way. Secondly, if \( c \) is defined over the variables of \( G \), the number of variables will always

---

actual suspension directly using techniques such as those proposed for analyzing programs with delay declarations (see Chapter 7).
be bounded, thus significantly simplifying the task of obtaining finite abstractions and finite abstract computations.

Let us now describe the concrete operational semantics (the standard semantics), on which the extended analysis framework is based. Consider a derivation of the state $s = \langle g_1, \ldots, g_n, c \rangle$ for a program $P$, which at some point has reached the state $s_i = \langle g_i, \ldots, g_n, c_i \rangle$. The new schema, based on the idea of replacing SLD by what is called LSLD (Local SLD) in [14], proceeds as follows:

- If $g_i$ is an atom, then for every rule $r : h \leftarrow B$ of the program $P$ such that $h$ and $a$ have the same predicate symbol, a local derivation is started as follows:

  1. If $\text{consistent}(c_i \land (g_i = h))$, then the state $s_{in} = \langle B, c_{in} \rangle$ is obtained by first computing $c_{entry} = \exists_{\text{var}(g_i)} c_i$ and then $c_{in} = \exists_{\text{var}(r)} (c_{entry} \land (g_i = h))$. Otherwise $s_{in} = \text{fail}$. This step is called procedure entry.

  2. The (local) derivation starting in $s_{in}$ is developed.

  3. For every final state $\langle \text{nil}, c_{out} \rangle$ resulting from the derivation starting in $s_{in}$, a state $s_{i+1} = \langle g_{i+1}, \ldots, g_n, c_{i+1} \rangle$ is derived by first computing $c_{exit} = \exists_{\text{var}(g_i)} (c_{out} \land (g_i = h))$ and then $c_{i+1} = c_i \land c_{exit}$. This step is called procedure exit. For every final state $\text{fail}$ resulting from the derivation starting in $s_{in}$, a state $s_{i+1} = \text{fail}$ is obtained. Otherwise, if all local derivations terminate without producing a final state, then the state $s_i$ does not have a successor state $s_{i+1}$.

- If $g_i$ is a constraint such that $\text{consistent}(g_i \land c_i)$ holds, then the state $s_{i+1} = \langle g_{i+1}, \ldots, g_n, c_{i+1} \rangle$ is directly obtained by computing $c_{i+1} = c_i \land g_i$. Otherwise ($\neg \text{consistent}(g_i \land c_i)$), the state $s_{i+1} = \text{fail}$.

Note that the original framework also considers the case in which $g_i$ is a builtin. Since this case can be directly carried over to the extended scheme it has been omitted for simplicity.

Notice that, under the standard semantics, $\exists_{\text{var}(g_i)} c_{i+1} = c_{exit}$. This is not necessarily the case after abstraction as will be discussed in Section 6.3.3. It is easy to see that the LSLD semantics is, under the given particularization, equivalent to the SLD semantics.

Within the proposed LSLD semantics, it is convenient to name constraints differently, depending on the point in a rule to which they correspond. The same conventions will be used for the abstract constraints. Consider, for example, the rule
Let $c_i$ and $c_{i+1}$ be the constraints to the left and right of the subgoal $b_i$, $1 \leq i \leq n$ in this rule. See Figure 6.1.

- $c_i$ and $c_{i+1}$ are, respectively, the call constraint and the success constraint for $b_i$.
- $c_1$ and $c_{n+1}$ are, respectively, the in constraint and the out constraint of the rule $r$ (also denoted by $c_{in}$ and $c_{out}$). Note that $c_1$ and $c_{n+1}$ are also the call constraint for $b_1$ and the success constraint for $b_n$, respectively.
- $c_i$ projected over the variables of $b_i$ is the entry constraint (represented by $c_{entry}$) of $b_i$ and the answer constraint for $b_i$ is called exit constraint (represented by $c_{exit}$). Note that these two constraints are defined over the variables in $b_i$, instead of over the variables of the rule.

### 6.3 Extension of the Analysis Framework

In this section we formalize the extension of the original analysis framework and provide safety conditions to be met by the user-defined functions. We will mainly follow the scheme of [92] in which a summary of the correctness conditions required in the original framework is given.

#### 6.3.1 The Abstract Domain

Let $\mathcal{R} = (\mathcal{D}, \mathcal{L})$ be a constraint domain defined as before. For each $\hat{x}$, we are interested in a concrete domain $(\text{Cons}_{\hat{x}}^C, \subseteq^C)$ where the elements of $\text{Cons}_{\hat{x}}^C$ (denoted $C$) are sets of satisfiable constraints\footnote{By abuse of notation we use the same notation $\hat{x}$ for a sequence of distinct variables and the corresponding set of variables.} over $\hat{x}$ and where $\subseteq^C$ is the subset relation. Notice that the concrete domain is a lattice, the minimal element is $\emptyset$ and the set of all constraints is the

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{constraint_names.png}
\caption{Constraint names}
\end{figure}
 maximal element. For convenience, the constraint *false* (representing an unsatisfiable constraint) may also occur in elements of the concrete domain.

The abstract domain $Cons_A$ consists of descriptions (denoted $AC$) and is equipped with an order relation $\leq^A$, $\alpha$ and $\gamma$ being, respectively, the abstraction and concretization function. In an abuse of notation we will sometimes denote the abstraction of a singleton $\{c\}$ in $Cons_A$ as $\alpha(c)$ instead of $\alpha(\{c\})$.

As stated in Section 2.3, the most familiar setting for abstract interpretation is the Galois insertion. The concretization function $\gamma$ and the abstraction function $\alpha$ provide a tight linkage between the concrete and the abstract domain. As a consequence one can specify the safety conditions of the functions used in formulating the abstract semantics either in terms of the concretization function or in terms of the abstraction function. As discussed in [111], abstract interpretation has also been studied in settings with a weaker linkage between the abstract and concrete domain. Here we follow the weaker setting of the original framework of Bruynooghe [10].

Given $(Cons_C, \leq^C)$ as the concrete domain (the powerset of the set of all constraints ordered by the subset relation) the requirements are:

1. A pre-order $\preceq^C$ satisfying that $\forall AC_1, AC_2 \in Cons_C$ if $AC_1 \preceq^C AC_2$ then $\gamma(AC_1) \subseteq \gamma(AC_2)$.
2. An equivalence relation $\approx^C$ satisfying that $\forall AC_1, AC_2 \in Cons_C$ if $AC_1 \approx^C AC_2$ then $\gamma(AC_1) = \gamma(AC_2)$.
3. An upper bound operator $\text{up} : Cons_C \times Cons_C \rightarrow Cons_C$ such that $AC_1 \preceq^C \text{up}(AC_1, AC_2)$ ($i = 1, 2$).
4. A maximal element named $\top$ such that $\gamma(\top) = \text{the set of all constraints and}$ $\forall AC \in Cons_C, AC \preceq^C \top$.
5. A minimal element $\perp$ such that $\gamma(\perp) = \emptyset$ and $\forall AC \in Cons_C, \perp \preceq^C AC$.
6. A subset $F_x$ of $Cons_C$ s.t. $F_x$ has no infinite ascending chain for $\preceq^C$, and $\top, \perp \in F_x$.
7. An operator $R : Cons \rightarrow F_x$ satisfying that $\forall AC \in Cons, AC \preceq^C R(AC)$.

According to conditions 1 and 2, it is allowed to have different descriptions for the same set of constraints, however the equivalence classes induced by the preorder have to be partially ordered. Condition 3 states that there must be an upper bound operator, i.e. that it must be possible to approximate two or more descriptions by a single one. Of course, it is desirable to define $\text{up}$ as precisely as possible. With the abstract domain being a complete partial order, the optimal $\text{up}$ is the least upper bound. Condition 4 implies the existence of a maximal element, it is a convention to name it $\top$. Condition
4 also states that such element must represent the set of all constraints. This ensures that every set of constraints has an abstraction. Condition 5 imposes a minimal element \( \bot \) representing the empty set of constraints. This provides a precise abstraction for states in unreachable program points. Also it provides the initial value for computing a fixpoint of a function over the abstract domain. Finally, conditions 6 and 7 guarantee the termination of the abstract interpretation procedure, needed when dealing with recursive predicates. Note that even considering these relaxed requirements, the adjoint nature of the abstraction and concretization functions remains unchanged.

6.3.2 The Abstract Operations

Consider a derivation starting in the abstract state \( as = \langle g_1 : \cdots : g_n, AC \rangle \) which, at some point, has reached the abstract state \( as_i = \langle g_1 : \cdots : g_n, AC_i \rangle \). Let us denote the abstract projection and conjunction functions by \( \exists^A \) and \( \wedge^A \), respectively. Then, if \( g_i \) is an atom:

- The \( \text{abstract}_\text{entry}(g_i, AC_i) \) function first obtains \( AC_{\text{entry}} = \exists^A_{\text{vars}(g_i)} AC_i \). Let \( r_1, \ldots, r_m \) be the rules of \( P \) (renamed apart with new variables) such that for all \( r_j : h_j \leftarrow B_j \), the call \( g_i \) and the head \( h_j \) have the same predicate symbol. Then it computes the abstract in constraint \( AC_{\text{in}}^j = \exists^A_{\text{vars}(r_j)} (AC_{\text{entry}} \wedge^A \alpha(g_i = h_j)) \) for each rule \( r_j \).

The safety of the \( \text{abstract}_\text{entry} \) function is ensured if:

- \( AC_{\text{in}}^j \) satisfies : for all \( c_i \in \gamma(AC_i) \) it holds that \( c_{\text{in}}^j \in \gamma(AC_{\text{in}}^j) \) where \( c_{\text{in}}^j = \exists_{\text{vars}(r_j)} (c_i \wedge (g_i = h_j)) \).
- \( AC_{\text{entry}} \in F_{\text{vars}(g_i)} \).

- Assume that the final state for the derivation of each abstract state \( as_i = \langle B_j, AC_{\text{in}}^j \rangle \) is \( \langle \text{nil}, AC_{\text{out}}^j \rangle \). Then \( \text{abstract}_\text{exit}(g_i, AC_i, \{h_1, \ldots, h_m\}, \{AC_{\text{out}}^1, \ldots, AC_{\text{out}}^m\}) \) first computes each \( AC_{\text{out}}^j = \exists^A_{\text{vars}(g_i)} (AC_{\text{out}}^i \wedge^A \alpha(g_i = h_j)) \). Next it computes the upper bound \( \text{upp} \) of \( \{AC_{\text{out}}^1, \ldots, AC_{\text{out}}^m\} \) obtaining \( AC_{\text{exit}} \). Finally it computes \( AC_{i+1} = AC_i \wedge^A AC_{\text{exit}} \) (this last step is called extension).

The safety of the \( \text{abstract}_\text{exit} \) function is ensured if:

- \( AC_{\text{exit}}^j \) satisfies : for all \( c_{\text{out}}^j \in \gamma(AC_{\text{out}}^j) \) it holds that \( c_{\text{exit}}^j \in \gamma(AC_{\text{exit}}^j) \) where \( c_{\text{exit}}^j = \exists_{\text{vars}(g_i)} (c_{\text{out}}^j \wedge (g_i = h_j)) \), and
- \( AC_{i+1} \) satisfies : for all \( c_i \in \gamma(AC_i) \) and \( c_{\text{exit}} \in \gamma(AC_{\text{exit}}) \) such that \( c_{\text{exit}} \rightarrow \exists_{\text{vars}(g_i)} (c_i) \) it holds that \( (c_i \wedge c_{\text{exit}}) \in \gamma(AC_{i+1}) \).
It is easy to note that the abstract functions have been defined by simply replacing the concrete conjunction and projection functions with their abstract counterparts. In fact, one can show that safety follows from the properties of \( \text{upp} \) and the safety of abstract projection and abstract conjunction. The latter can be formulated as follows:

- The abstract projection \( \exists_A x \) is a safe approximation of the concrete projection if for any constraint \( c \) and for any abstract constraint \( AC \) such that \( c \in \gamma(AC) \) it holds that \( \exists_A x \in \gamma(\exists_A x AC) \).

- The abstract conjunction \( \land A \) is a safe approximation of the concrete conjunction if for any two constraints \( c_1, c_2 \) and for any two abstract constraints \( AC_1, AC_2 \) such that \( c_1 \in \gamma(AC_1) \) and \( c_2 \in \gamma(AC_2) \) it holds that \( c_1 \land c_2 \in \gamma(AC_1 \land A AC_2) \).

We now have all the elements to describe the kernel of the analysis framework: the \text{call\_to\_success}(g_i, AC_i) \) procedure which computes the abstract success constraint \( AC_{i+1} \) of the subgoal \( g_i \) from its abstract call constraint \( AC_i \). If \( g_i \) is a constraint, then the abstract success constraint \( AC_{i+1} \) is obtained by simply applying the abstract conjunction function of \( g_i \) and \( AC_i \). If \( g_i \) is an atom, it will proceed with the following steps:

1. Compute \( AC_{\text{entry}} = \exists_A x_{\text{var}(g_i)} AC_i \).

2. If \( g_i \) has no ancestor node for an identical (up to renaming) \( g_i' \) with an abstract projected constraint equivalent to \( AC_{\text{entry}} \):

   2.1 Apply \text{abstract\_entry}(g_i, AC_i) \) obtaining the set of abstract entry constraints \( AC_{\text{entry}}^1, \ldots, AC_{\text{entry}}^m \), one for each rule \( h_j \leftarrow B_j \).

   2.2 The states \( (B_j, AC_{\text{entry}}^m) \) are analyzed, applying, from left to right, \text{call\_to\_success} to the subgoals of the \( B_j \). Eventually one obtains the abstract out constraints \( AC_{\text{out}}^1, \ldots, AC_{\text{out}}^m \).

   2.3 Apply \text{abstract\_exit}(g_i, AC_i, \{h_1, \ldots, h_m\}, \{AC_{\text{out}}^1, \ldots, AC_{\text{out}}^m\}) \) obtaining \( AC_{i+1} \).

3. If \( g_i \) has an ancestor node for an identical (up to renaming) \( g_i' \) with an abstract projected constraint \( AC_{\text{entry}}' \) equivalent to \( AC_{\text{entry}} \), then apply \text{abstract\_exit}(g_i, AC_i, \{g_i'\}, \{AC_{\text{exit}}'\}) \) obtaining the abstract success constraint \( AC_{i+1} \). At some point a new abstract exit constraint for \( g_i' \) will be computed. If the new abstract exit constraint is less or equal than the previous one, the computation will proceed normally. Otherwise, the computation for \( g_i' \) is restarted.

\footnote{Here, and in what follows, we implicitly assume proper renaming of formulas.}
using the upper bound between the previous and the new abstract exit constraint as the new value for $AC_{\text{exit}}$.

It is important to note that in the original analysis framework it is possible to change step 1 in case the depth of recursion is beyond a predefined threshold, thus performing a kind of widening [41]. Since this option has no incidence on the correctness of the approach, it has been omitted for reasons of clarity. An extended version of the framework presented including this feature can be found in [45].

### 6.3.3 Improving the abstract \textit{exit} Function

There are some domain-dependent issues which can significantly affect the precision of the results obtained by the \textit{abstract \textit{exit}} function. In this section we will discuss two of such issues.

The first issue is related to the straightforward extensions of the three properties identified by Jacobs and Langen [85]: \textit{idempotence}, \textit{commutativity} and \textit{additivity}. Idempotence implies that repeating abstract conjunction does not change the result. Commutativity allows abstract conjunction to be performed in any order. Finally, additivity guarantees that precision is not lost when performing least upper bounds. Idempotence is satisfied for most of the abstractions proposed in the literature; for example, the abstract conjunction functions of the analyzers based on the ASub, Sharing, and Sharing+Freeness domains evaluated in the previous chapter, are idempotent. Commutativity is satisfied for quite a few abstractions; for example, although the abstract conjunction function of the analyzer based on the ASub domain is not commutative, those defined for the Sharing and Sharing+Freeness are commutative. Unfortunately, additivity is not common, one significant exception is the upper bound function defined by Marriott and Søndergaard for the \textit{Prop} domain [107].

It is clear that if the upper bound is not additive, obtaining $AC_{i+1}$ by computing first $AC_{\text{exit}}$ as the upper bound $\{AC_{\text{exit}}^1, \ldots, AC_{\text{exit}}^m\}$ and then obtaining $AC_{i+1} = AC_i \land^A AC_{\text{exit}}$, as it is performed by the \textit{abstract \textit{exit}} function may imply a loss of precision w.r.t. the function which first obtains each $AC_{i+1}^j = AC_i \land^A AC_{\text{exit}}^j$ and then obtains $AC_{i+1}$ as the upper bound of $\{AC_{i+1}^1, \ldots, AC_{i+1}^m\}$.

**Example 6.3.1** Consider the abstract domain \textit{Def} [6] in which each abstract constraint $AC_x$ is a Boolean function of the form

$$ \bigwedge_{x \in \mathbb{D}} (x \leftarrow M_x) $$

138
where $M_x$ is monotonic. When abstracting definiteness, each implication approximates the conditions under which $x$ becomes definite. For example, the formula

$$x \leftarrow ( (y \land z) \lor u ) \land y \leftarrow \text{false} \land z \leftarrow \text{false} \land u \leftarrow \text{false} \land v$$

approximates the concrete constraints in which $v$ is definite, $x$ becomes definite if either both $y$ and $z$, or $u$ become definite, and nothing is known about the definiteness of $y$, $z$ and $u$.

Let $AC_i = \{ x \leftarrow y \lor z \}$, $AC^1_{exit} = \{ y \}$, and $AC^2_{exit} = \{ z \}$. While in the former approach, $AC_{exit} = \text{upp}(\{ AC^1_{exit}, AC^2_{exit} \}) = \text{true}$ and thus after the abstract conjunction we obtain $AC_{i+1} = \{ x \leftarrow y \lor z \}$, in the latter we first compute $AC^1_{i+1} = \{ x \land y \}$ and $AC^2_{i+1} = \{ x \land z \}$, thus the upper bound being $AC_{i+1} = \{ x \}$.

Thus, for most of the abstract domains and abstract functions proposed in the literature, the modified abstract $\text{exit}$ function may yield a more accurate result. Unfortunately, the abstract conjunction is, in general, quite expensive. Since the change proposed may significantly increase the number of times in which this function is applied, changing or not the abstract $\text{exit}$ function is a question of an accuracy/efficiency tradeoff.

The second issue is related to the interaction among non normalization, a loss of precision in the abstract projection function, and the tabulation method. Let us first discuss the tabulation method. As explained in [10], bundling together the states in the standard semantics into a collecting state allows to organize the computation in an AND-OR tree. This computation schema can be extended with tabulation, yielding LSLDT [14] in the same way as OLD can be extended with tabulation, yielding OLDT [145]. Systems such as PLAI [121, 122, 125] and GAIA [150, 7] store the call $g_t$ and the associated abstract projected constraint $AC_{entry}$ as table entries. The corresponding "answers" are the abstract exit constraints $AC_{exit}$ obtained by the abstract $\text{exit}$ function. Let us now introduce the problem with a simple example.

**Example 6.3.2** Consider the Herbrand constraint domain and a simple abstract domain in which definiteness information is described by means of the set of program variables $G$ which are known to be constrained to a unique value (the first component of the $\text{ASub}$ domain described in the previous chapter).

Assume we have a program $P$ with only one clause $r : p(z)$. (i.e., a fact) and the query to the analysis is $\text{call fail success}(p(f(x,y)), AC_{i})$, where $AC_1 = \{ x \}$ (i.e., the program variable $x$ will be (at run-time) constrained to a unique value any time that $p(f(x,y))$ is called). Following the previous abstract framework we would perform the following steps:
1. \( \text{abstract}_{\text{entry}}(p(f(x,y)), AC_1) \). We obtain the abstract projected constraint \( AC_{\text{entry}} = \{x\} \). Then we abstract the constraint \( p(f(x,y)) = p(z) \) which results in \( \emptyset \), and perform the abstract conjunction with \( AC_{\text{entry}} \) obtaining \( \{x\} \). Finally, we project it over the variables in \( p(z) \) obtaining \( AC_{\text{in}} = \emptyset \).

2. Since the body of \( r \) is empty, \( AC_{\text{out}} = AC_{\text{in}} = \emptyset \).

3. \( \text{abstract}_{\text{exit}}(p(f(x,y)), AC_1, \{p(z)\}, \{AC_{\text{out}}\}) \). We abstract the constraint \( p(f(x,y)) = p(z) \) which results in \( \emptyset \), then we abstractly conjunct it with \( AC_{\text{out}} = \emptyset \) obtaining \( \emptyset \) and then we will abstractly project it onto the variables in \( p(f(x,y)) \) obtaining \( AC_{\text{exit}} = \emptyset \). Finally, we will perform the abstract conjunction of \( AC_1 \) and \( AC_{\text{exit}} \) resulting in \( AC_2 = \{x\} \). □

In the example we can observe that the definiteness of \( x \) is lost in \( AC_{\text{exit}} \) and recovered in the abstract success constraint \( AC_2 \). The reason for this loss is that the information about the definiteness of the first argument of the term to which \( z \) is bound is lost during the abstract projection performed for obtaining the abstract entry constraint \( AC_{\text{in}} \). Thus in the abstract exit function, the definiteness of \( x \) cannot be inferred, and it will be lost until performing the abstract conjunction with \( AC_1 \). One could think that this loss is not relevant since the obtained \( AC_{\text{exit}} \) will always be safe and the accuracy will be recovered as soon as the abstract success constraint \( AC_2 \) is computed. Furthermore, if a successor node uses the information contained in \( AC_{\text{exit}} \) there would be no loss of accuracy. The reason is that in order to use the information in \( AC_{\text{exit}} \) the abstract projected constraint for \( g_i \) in the successor node must be equivalent to (a variant of) \( AC_{\text{entry}} \) and therefore the accuracy will also be recovered when computing the abstract success constraint \( AC_2' \) for the successor node.

However, there are many practical reasons which recommend not to allow this behavior. The main reason is that, if such \( AC_{\text{exit}} \) is stored as an “answer” of the associated table entry (as it is usually done), it could imply a greater number of fixpoint iterations. Recall that a new fixpoint iteration will be started any time that the new computed value for \( AC_{\text{exit}} \) is not less or equal than the previous one. Also, there are many abstract domains whose efficiency greatly depends on the level of accuracy, and thus obtaining more compact abstractions can increase the performance. Finally, many applications are based on the information given by the abstract projected and abstract exit constraints, such as in mode-inferencing applications. Then, before making any decision based on the abstract exit constraint, an abstract conjunction between the abstract projected and exit constraints has to be performed.

Most of these problems can be avoided by storing the abstract constraint resulting of projecting \( AC_{i+1} \) onto the variables of the goal, instead of the \( AC_{\text{exit}} \) computed by
the abstract\_exit function. However, we argue that, in general, it is more efficient to re-define the abstract exit function in such a way that this loss would be avoided. The definition is the following. Let $g_i$, $AC_i$, $AC_{proj}$, $\{h_1, \cdots, h_m\}$, and $\{AC^1_{out}, \cdots, AC^m_{out}\}$ be as before:

The $\text{abstract\_exit}(g_i, AC_i, AC_{entry}, \{h_1, \cdots, h_m\}, \{AC^1_{out}, \cdots, AC^m_{out}\})$ function will first compute each $AC^j_{exit} = A_{\text{var}(g_i)}(AC^j_{out} \land A (AC_{entry}) \land A (g_i = h_j))$.

Then it will compute the upper bound of $\{AC^1_{exit}, \cdots, AC^m_{exit}\}$ obtaining $AC_{exit}$.

Finally it will compute $AC_{i+1} = AC_i \land A AC_{exit}$.

It is easy to see that the only modification w.r.t. the previous definition of the abstract\_exit function is the inclusion of the abstract projected constraint $AC_{entry}$ in the computation of the $AC^j_{exit}$. We argue that this solution is better for three main reasons. Firstly, the abstract constraint $AC_{entry} \land A (g_i = h_j)$ is already computed during the abstract\_entry function. Thus it can be reused in the abstract\_exit function avoiding a performance impact. Secondly, since the intermediate abstract constraints obtained in the process of computing $AC_{i+1}$ may be more accurate, and thus more “compact”, this can increase the performance in domains which are sensitive to the number of variables involved. Finally, if $AC_{entry}$ has been included in the computation of $AC_{exit}$, conjuncting $AC_{exit}$ with $AC_i$ only implies propagating the new information to the variables not in $g_i$, thus possibly avoiding many operations.

### 6.3.4 Passive Constraints

The extended analysis framework as defined above does not consider passive constraints. This restriction has been assumed in order to follow as close as possible the definition of the original analysis framework which did not allow for passive constraints. Integrating passive constraints can be done by returning to the original representation of a state as a tuple $\langle G, c, s \rangle$ (but $G$ being a sequence rather than a multi-set), performing the standard conjunction operation on $s$ instead of on $c$, and adding an $\text{infer}(c, s) = (c', s')$ step after each conjunction operation.

The fundamental question is what kind of information is required from the analysis and at what level of accuracy. If no information regarding which constraints are passive and when they become active is required, and we prefer to lose accuracy rather than complicate the abstract operations, the simplest method is to abstract both active and passive constraints by a single abstract component, without distinguishing between the information regarding passive constraints and that provided by the active constraints. This abstraction has to be safe with respect to all possible (future) activations of the passive constraint (explained further below), and therefore it is possible to lose accuracy.
However, this method significantly simplifies the abstract operations and allows such analysis to be integrated in the framework described above. We adopted this simple approach in the implementation of the analyzer presented in Section 6.4.

If the information provided by the analysis is aimed at detecting program points at which all constraints are active, then we have to abstract in some way the $\text{infer}$ function. In order to do this, the abstract domain should be able to approximate the information used for $\text{infer}$ to decide if a constraint is definitely active. Then, for each constraint $c$ analyzed, the abstract $\text{infer}$ function must decide if under the current abstract constraint store, $c$ is definitely active, and, if it is not the case, it must abstract the fact that a passive constraint may appear (possibly without identifying which particular constraint it is) and the properties needed for such passive constraint to be definitely woken. Note that, if the information abstracted by the domain is downwards closed and we are not interested in call patterns, then there is no need to deal with possible wake-ups of passive constraints. This option is closely related to the work presented in [67] which presents an abstract domain for detecting CLP($\mathcal{R}$) programs for which all passive non-linear constraints eventually become linear at runtime. Otherwise, we must take possible wake-ups into account.

Finally, if the information is aimed at accurately modeling the delay and wake-up behavior, and we want to be able to determine which are the passive constraints, when they become passive and when they are woken, we should split up the abstraction in two parts, an active part representing the active constraints and a passive part representing the passive constraints. In this case, the abstract projection function has to preserve enough information to ensure the correct wake-up behavior. A possible technique is to project only the abstract active constraints and to keep the passive part. In that case, an abstract constraint is no longer restricted to a finite number of variables (the variables of the rule, goal or query) as it is in the original abstract interpretation framework. As a consequence, termination is not guaranteed and some new kind of widening should be introduced. This is related to the work described in the next chapter which gives a simple denotational semantics and a generic global data-flow analysis algorithm which is based on the semantics sketched above, for languages in which computation generally proceeds left-to-right but in which some calls are dynamically delayed until their arguments are sufficiently instantiated, a very similar case to that of the passive constraints.
6.4 Inference of Definiteness Information

In this section we will present an abstract domain for the inference of definiteness information in CLP programs and the corresponding abstract functions as required for the extended framework developed above. The abstraction is based on a high-level description of uniquely constraining patterns which is then easy to obtain for each particular type of constraint in an actual system.

There are at least two other domains similar to ours which have been recently (and independently) proposed. One is the domain proposed by Hanus in [67] for detecting situations in which residuation\(^6\) does not need to be invoked. The author forces all the arguments of a residual function to be bound to ground terms in order to allow the evaluation and uses a domain similar to the one defined in this section for inferring groundness.

The second related domain, briefly introduced in Example 6.3.1, is the domain of positive Boolean functions which are closed under intersection. This domain was defined early on by Dart [43] under the name of dependency formulae and applied to the inference of groundness in deductive databases. Our domain can be seen as a compact representation of this domain (including a formulation of efficient operations for it). The main difference is that, for efficiency reasons, we require the abstraction to be in a particular solved form. Recently, the different possible subsets of the boolean functions which can be used for tracking dependencies in program analysis and their representations have been studied and greatly clarified [6]. Our domain corresponds essentially to the \textit{Def} domain in this taxonomy. The work developed in [6] also illustrates that the representation that we propose is closely related to the \textit{CDF} representation which is shown therein to offer an advantageous cost-performance tradeoff.

We will briefly summarize herein some of the results presented in [6] and discuss the differences between the two domains. The motivation behind this is to give a better intuition of our domain and simplify the proofs of correctness of the abstract functions by using the domain \textit{Def} and the results for it. Previously, in [44], we established a similar relation with the domain \textit{Prop} defined by Marriott and Søndergaard, with the same objective. However, it seems that \textit{Def} is much closer to our domain than \textit{Prop}, and thus it is better to use \textit{Def} as the reference point.

\textit{Def} can be defined as the set of functions which can be written in the form:

\[
\bigwedge_{x \in \text{Var}} (x \leftarrow M_x)
\]

\(^6\)Residuation is an operational mechanism for the integration of functions into logic programming. The residuation principle delays the evaluation of functions during the unification process until the arguments are sufficiently instantiated.
where $M_x$ is monotonic. Following the notation of [6] and [1], we will denote $Mon$ as the set of monotonic Boolean functions, and we will say $F = \Omega S$ to indicate that the set $S$ of connectives suffices, together with variables, to represent every function in $F$, and no function outside of $F$ can be represented. For example, it is well known that $Mon = \Omega \{ \land, \lor \}$. In this context $Def$ is a proper subset of $Prop$ (referred to as $Pos$ in their work). Furthermore, the monotonic $M_x$ functions mentioned above can be written in such a way that $x$ does not appear in $M_x$. For example,

$$(x \leftarrow y \land z) \land (x \leftarrow u) \land v$$

can be written as

$$x \leftarrow ((y \land z) \lor u) \land y \leftarrow false \land z \leftarrow false \land u \leftarrow false \land v$$

When abstracting definiteness, the meaning of the above formula would be as follows, $v$ is definite, $x$ becomes definite if either both $y$ and $z$, or $u$ become definite, and nothing is known about the definiteness of $y, z$ and $u$. Our approximation of definiteness is similar to this. Informally, an abstract constraint $AC^D = (D, R)$ of the abstract domain $Cons^D$ consists of two elements. The first element $D$ approximates the set of variables which are known to be definite, i.e., those which appear as an assertion in the above formula. The second element $R$ approximates definite dependencies. $R$ is a set of couples $(x, SS)$ where $SS$ is a set of set of variables. It is clear that the couple approximates the implication connective, and $SS$ approximates the monotonic function in disjunctive normal form. For example, the above formula would be represented as $((v), ((x, \{ y, z \}, \{ u \})).$ Note that variables for which nothing is known (such as $y, z$ and $u$) do not appear in the abstraction.

The main differences between the two domains are that we always keep the abstractions in a particular solved form. The objective is both to obtain a more compact representation and to reduce the cost of key operations, such as testing for equivalence of two abstractions and performing abstract projection. The improvements are due to the propagation of the definiteness information until (a) all variables known to be definite appear in $D$, and (b) for any $(x, SS) \in R$, $SS$ is the minimum set of sets containing all possible definiteness relationships affecting $x$. For example, the formula

$$x \leftarrow y \land y \leftarrow z \land z \leftarrow false$$

in $Def$, must be expressed in $Cons^D$ by $(\emptyset, \{(x, \{ y \}, \{ z \}), (y, \{ \{ z \} \}))$. With such representation, the tests for equivalence of two abstractions become a test for identity, and the projection function is greatly simplified.
6.4.1 Abstract Domain and Abstraction Function

Let us first define some simple functions which will be used to eliminate unnecessary complexity from the abstract conjunction function. The function $\text{min}$ simplifies the dependency information inferred for a given variable by eliminating supersets. In the $\text{Def}$ domain, this would imply replacing the monotonic function $((y \land z) \lor y)$ in the implication $x \leftarrow ((y \land z) \lor y)$ with the monotonic function $y$. This corresponds to putting the $SS$ components in what is called $\text{RDNF}$ (reduced disjunctive normal form) in $\land[6]/\lor$, which is obtained by eliminating the terms (or sets) in $SS$ which are implied by other terms, thus eliminating redundancies. Formally, let $\text{Var}$ denote an enumerable set of variables and $P\text{var} \subseteq \text{Var}$ a distinguished (enumerable) set of variables which may occur in programs. Let $\wp(S)$ denote the powerset of a set $S$.

**Definition 6.4.1**

Let $SS \in \wp(\wp(\text{Pvar}))$. Then $\text{min}(SS) = \{S \in SS \mid \exists S' \in SS, S' \subset S\}$

The function $\text{prop}_gr$ recursively propagates the definiteness of a set of variables $V$ to a (possibly not in solved form) abstract constraint $(D,R)$.

**Definition 6.4.2**

Let $V, D \in \wp(\wp(\text{Pvar}), R \in \wp(\wp(\text{Pvar} \times \wp(\wp(\text{Pvar}))))$. Then

$$\text{prop}_gr(V, (D, R)) = \begin{cases} \text{if } V = \emptyset & \text{then } (D, R) \\ \text{else } \text{prop}_gr(D', (D \cup V \cup D', R')) \end{cases}$$

where

$D' = \{x \mid (x, SS) \in R, x \in (V \cup D)\} \cup \{x \mid (x, SS) \in R, \exists S \in SS, S \subseteq (V \cup D)\}$

$R' = \{(x, \text{min}(\{S \setminus (V \cup D) \mid S \in SS\}) \mid (x, SS) \in R, x \notin D'\}$

**Example 6.4.1**

$$\text{prop}_gr(\{x\}, (\emptyset, \{(y, \{\{x\}\}), (z, \{\{x, w\}\})\})) = \text{prop}_gr(\{y\}, (\{x, y\}, \{\{z, \{\{w\}\}\}\})) = \text{prop}_gr(\emptyset, (\{x, y\}, \{z, \{\{w\}\}\})) = (\{x, y\}, \{z, \{\{w\}\}\})$$

The function $\text{prop}_dep$ recursively propagates the dependency information contained in each element of the (possibly not in solved form) dependency element $R$ of an abstract constraint. Neither $\text{prop}_dep$ nor $\text{prop}_gr$ are applied to the elements of $\text{Def}$.

**Definition 6.4.3**

Let $R, R_1, R_2 \in \wp(\text{Pvar} \times \wp(\wp(\text{Pvar})))$. Then $\text{prop}_dep(R) = \text{prop}_dep(R, R)$, where
\[\text{prop}\_\text{dep}(R_1, R_2) = \begin{cases} \text{if } R_1 = \emptyset \text{ then } R_2 \\ \text{else let } (x, SS) \in R_1 : \text{prop}\_\text{dep}(R_3, R_4) \end{cases}\]

where
1. \( R_4 = \{(y, \min(SS_2 \cup SS')) | (y, SS_2) \in R_2, SS' = \{S_2 \cup S | ((x) \cup S_2) \in SS_2, S \in SS, y \notin S\}\}\)
2. \( \text{Changed} = \{(y, SS_1) | (y, SS) \in R_4, (y, SS_2) \in R_2, SS_2 \neq SS_1\}\)
3. \( R_3 = \text{Changed} \cup \{(y, SS_1) | (y, SS_1) \in R_1, y \neq x, \neg \exists(y, SS_2) \in \text{Changed}\}\)

First we compute \( R_4 \) based on the following idea. If the definiteness of \( x \) can be inferred from that of the variables in \( S \subseteq SS \), and the definiteness of some \( y \) can in turn be inferred from that of \( \{x\} \cup S_2 \), we can conclude that the definiteness of \( y \) can also be inferred from that of \( S_2 \cup S \). Then we compute \( \text{Changed} \), i.e., the set of elements in \( R_4 \) which have changed w.r.t. \( R_2 \) and thus their information has to be propagated again. Finally, we compute \( R_3 \) as the union of \( \text{Changed} \) and the rest of the elements in \( R_1 \), whose information has not changed due to that of \((x, SS)\).

**Example 6.4.2**

\[
\text{prop}\_\text{dep}([[x, \{y, z\}]], (y, \{z\})), (x, \{y, z\}), (y, \{z\})) = \text{prop}\_\text{dep}((y, \{z\}), (x, \{y, z\}), (y, \{z\})) = \\
\text{prop}\_\text{dep}((x, \{z\}), (x, \{z\}), (y, \{z\})) = \\
\text{prop}\_\text{dep}(\emptyset, (y, \{z\}), (x, \{z\})) = \\
(y, \{z\}), (x, \{z\}) \square
\]

We can now formally define \( \text{Cons}^D \) as \( \bot \cup \{(D, R) \in \psi(P\text{var}) \times \psi(P\text{var} \times \psi(P\text{var}))\}^7 \) such that:

1. \( \text{prop}\_\text{pr}(D, R) = (D, R) \)
2. \( \forall (x, SS) \in R, SS \neq \emptyset, SS \neq \emptyset \)
3. \( \text{prop}\_\text{dep}(R) = R \)

The above conditions ensure that the information contained in each abstract constraint is coherent, i.e., (1) the definiteness of each variable in \( D \) has been already propagated to \( R \), (2) the dependency information (if any) for each possible non definite variable is meaningful, and (3) it has been already propagated to the rest of elements in \( R \).

We say that an element of \( \psi(P\text{var}) \times \psi(P\text{var} \times \psi(P\text{var})) \) is an "intermediate" abstract constraint if it satisfies (2) but possibly neither (1) nor (3). They will appear in the intermediate steps performed during the abstract conjunction operation.

---

7For reasons of readability most of the following definitions and operations do not explicitly deal with \( \bot \). Their extensions are trivial.
Definition 6.4.4 [Abstraction of a constraint: \( \alpha^d \)]
Let \( c \) be an constraint.
Then \( \alpha^D(c) = \bot \) if \( \neg \text{consistent}(c) \), otherwise \( \alpha^d(c) = (D, R) \) where
1. \( D = \text{def}(c) \)\(^8\)
2. \( R = \{ (x, SS) \mid x \in \text{vars}(c), SS = \min(\text{grdep}(c, x)), SS \neq \emptyset, SS \neq \emptyset \} \)
3. \( \text{grdep}(c, x) = \{ y \subseteq \text{vars}(c) \setminus \{ x \} \mid d' \in \text{Cons}^G \text{ s.t. } \text{consistent}(c \land d') \text{ and } y \subseteq \text{def}(d'), \text{ then } x \in \text{def}(c \land d') \} \)

Note that \( \emptyset \in \text{grdep}(c, x) \) for any \( x \in \text{def}(c) \), and thus \( \min(\text{grdep}(c, x)) = \{ \emptyset \} \).
Also \( \text{grdep}(c, x) = \emptyset \) if even if all variables in \( c \) but \( x \) are definite, \( x \) is still not definite, i.e., when the definiteness of \( x \) does not depend on the definiteness of any set of variables.

Example 6.4.3
\[
\begin{align*}
\alpha^D(x = f(y, z)) &= \{ \emptyset, (x, \{ \{y, z\}\}), (y, \{\{x\}\}), (z, \{\{x\}\}) \} \\
\alpha^D(x = 3y + 2z) &= \{ \emptyset, (x, \{\{y, z\}\}), (y, \{\{x\}\}), (z, \{\{x, y\}\}) \} \\
\alpha^D(x > y) &= \emptyset, (\emptyset, \emptyset) \\
\alpha^D(x \neq y) &= \emptyset, (\emptyset, \emptyset) \\
\alpha^D(x =< y > z) &= \{ \emptyset, (x, \{\{y, z\}\}), (y, \{\{x\}\}), (z, \{\{x\}\}) \} \\
\alpha^D(x =< y > w > z) &= \{ \emptyset, (x, \{\{y, w, z\}\}), (y, \{\{x\}\}), (w, \{\{x\}\}), (z, \{\{x\}\}) \} \\
\alpha^D(x =< y > w, z) &= \{ \emptyset, (x, \{\{y, w, z\}\}), (y, \{\{x\}\}), (w, \{\{x, z\}\}), (z, \{\{x, w\}\}) \} \Box
\end{align*}
\]
Note that the symbol “.” stands for concatenation of lists, and “\(< y >\)” is a list with one element.

Definition 6.4.5 [Order relation]
Let \( (D_1, R_1), (D_2, R_2) \in \text{Cons}^D \).
Then \( (D_1, R_1) \leq^D (D_2, R_2) \) iff:
1. \( D_2 \subseteq D_1 \)
2. \( \forall (x, SS_2) \in R_2, x \in D_1 \) or \( \exists (x, SS_1) \in R_1 \) s.t. \( \forall S_2 \in SS_2, \exists S_1 \in SS_1, S_1 \subseteq S_2 \)

It is straightforward to see that the order function is satisfied if for every property in \( (D_2, R_2) \), there holds a property in \( (D_1, R_1) \) which is at least as strong.

\(^8\)Recall that \( \text{def}(c) \) denotes the set of definite variables in \( c \).
**Definition 6.4.6** [Equivalence]
Let \((D_1, R_1), (D_2, R_2) \in Cons^D\).
Then \((D_1, R_1) \equiv^D (D_2, R_2)\) iff:
1. \(D_1 = D_2\)
2. \(R_1 = R_2\)

**Definition 6.4.7** [Least upper bound]
Let \((D_1, R_1), (D_2, R_2) \in Cons^D\).
Then \(upp^D ((D_1, R_1), (D_2, R_2)) = (D, R)\) where:
1. \(D = D_1 \cap D_2\)
2. \(R = \{(x, SS) \in R_i \mid x \in D_j, i, j \in \{1, 2\}, i \neq j\} \cup \{(x, \min(SS')) | SS' = \{S_1 \cup S_2 | (x, SS_1) \in R_1, S_1 \in SS_1, (x, SS_2) \in R_2, S_2 \in SS_2\}\)

The definition is equivalent to the least upper bound in Def and it can easily be extended to compute the least upper bound of \(m (m > 2)\) abstractions. In the following we will assume that the function \(upp\) applies to a set of abstract constraints.

**Definition 6.4.8** [Abstraction of a set of constraints: \(\alpha^D\)]
Let \(C \in Cons^C\).
Then \(\alpha^D(C) = \bot\) if \(C = \emptyset\), otherwise \(\alpha^D(C) = upp((\alpha^d(c) \mid c \in C))\).

**Definition 6.4.9** [Maximal and minimal elements]
The maximal element is \(\top = (\emptyset, \emptyset)\).
The minimal element is \(\bot\), denoting the empty set of constraints.

The concretization function \(\gamma^D\) can be defined based upon \(\alpha^D\) as described in [41] :
\(\gamma^D(AC) = \bigcup\{C \in Cons^C \mid \alpha^D(C) \leq^D AC\}\). Then \((Cons^C, \subseteq, Cons^D, \leq^D)\) is a Galois insertion.

### 6.4.2 Abstract Projection and Abstract Conjunction Functions

**Definition 6.4.10** [Abstract projection]
Let \((D_1, R_1) \in Cons^D\) and \(\tilde{x}\) be a set of variables.
Then \(\pi^D_{\tilde{x}}(D_1, R_1) = (D, R)\) where:
1. \(D = D_1 \cap \tilde{x}\)
2. \(R = \{(x, SS) \mid (x, SS_1) \in R_1, SS = \{S \in SS_1 \mid S \subseteq \tilde{x}, SS \neq \emptyset\}\}

Intuitively, \(D\) is the subset of variables in \(\tilde{x}\) which are known to be definite in \(D_1\), and \(R\) contains the definiteness dependencies (if any) approximated by \(R_1\) for the possible non definite variables in \(\tilde{x}\). Since only the variables in \(\tilde{x}\) are taken into account,
any element \((y, SS_1) \in R_1\) approximating the dependencies for a variable which is not in \(\hat{x}\) (i.e., \(y \notin \hat{x}\)) is eliminated. Furthermore, the dependency sets \(SS_1\) of the elements \((x, SS_1) \in R_1, x \in \hat{x}\) which are not subsets of \(\hat{x}\) are also eliminated as groundness of all variables in a dependency set is required to ground \(x\), yielding \(SS\). Note that if as a result \(SS\) becomes empty, there is no information for the definiteness dependencies of \(x\) w.r.t. the variables in \(\hat{x}\) and no \((x, SS)\) will appear in \(R\).

**Definition 6.4.11** [Abstract conjunction]

Let \((D_1, R_1), (D_2, R_2) \in Cons^D\).

Then \((D_1, R_1) \land^D (D_2, R_2) = (D, R)\) where

1. \((D, R') = prop_{\text{dep}} ((D_1 \setminus D_2) \cup (D_2 \setminus D_1), (D_1 \cup D_2, R''))\)

2. \(R'' = \{(x, \min (SS_1 \cup SS_2)) \mid (x, SS_2) \in R_2, (x, SS_1) \in R_1 \} \cup \{(x, SS_1) \in R_1 \mid \exists (x, SS_2) \in R_2, x \notin D_2\} \cup \{(x, SS_2) \in R_2 \mid \exists (x, SS_1) \in R_1, x \notin D_1\}\)

3. \(R = prop_{\text{dep}}(R')\).

Informally, the \(\delta \land D\delta\) function proceeds in three steps. First it computes \((D_1 \cup D_2, R'')\), which abstracts the conjunction of the constraints before performing a simplification of the resulting abstract constraint. This is equivalent to conjunction in Def. Then it propagates the definiteness of the new set of definite variables obtaining \((D, R')\). The last step consists of propagating the dependency information of each dependency element until fixpoint is reached.

**Example 6.4.4**

Consider the abstract constraints:

\[
\begin{array}{|c|c|}
\hline
(D_1, R_1) & (\emptyset, \{(x, \{y, z\}), (y, \{z\})\}) \\
(D_2, R_2) & (\{z\}, \{y, \{w\}\}, \{w, \{y\}\}) \\
\hline
\end{array}
\]

Then \((D_1, R_1) \land^D (D_2, R_2)\) obtains the abstract constraint \((D_1 \cup D_2 \cup D, R)\) as follows:

1. \(R'' = \{(x, \{y, \{z\}\}), (y, \{z\}, \{w, \{y\}\})\}\).

2. \((D, R') = \text{prop}_{\text{dep}} ([\{z\}, \{y, \{w\}\}, \{w, \{y\}\}) = \text{prop}_{\text{dep}}([x,y], \{x,y,z\}, \{w, \{y\}\}) = \text{prop}_{\text{dep}}([w], \{x,y,z,w\}, \emptyset) = \text{prop}_{\text{dep}}(\emptyset, \{x,y,z,w\}, \emptyset) = \{(x, y, z, w, \emptyset)\}\)

3. \(R = \text{prop}_{\text{dep}}(R') = \text{prop}_{\text{dep}}(\emptyset) = \emptyset\)

Thus \((D_1, R_1) \land^D (D_2, R_2) = \{(x, y, z, w, \emptyset)\}\)

Consider now the abstract constraints:
Then \((D_1, R_1) \land^D (D_2, R_2)\) obtains the abstract constraint \((D_1 \cup D_2 \cup D, R)\) as follows:

1. \(R' = \{(x, \{y\}, \{z, w\}), (y, \{\{z\}\}, (z, \{\{y\}\})\}\)

2. \((D, R') = \text{prop}(\emptyset, (\emptyset, R')) = (\emptyset, R')\)

3. \(R = \text{prop}(\{(x, \{y\}, \{z, w\}), (y, \{\{z\}\}), (z, \{\{y\}\}), (x, \{\{y\}, \{z, w\}\}, (y, \{\{z\}\}), (z, \{\{y\}\})\})\)

Thus \((D_1, R_1) \land^D (D_2, R_2) = (\emptyset, \{(x, \{y\}, \{z\}), (y, \{\{z\}\}), (z, \{\{y\}\})\})\)

\(\Box\)

The rest of the abstract functions required by the framework (abstract entry and abstract exit) can be defined by simply replacing the general abstract functions used in the definitions given in Section 6.3.2 with those belonging to the definiteness domain.

6.5 Experimental Results

The abstract domain described in the previous section has been implemented within the PLAI abstract interpretation system, briefly described in Section 5.3.1. As mentioned, from the system point of view, it is sufficient to specify the particular abstract domain desired. This information is passed to the fixpoint algorithm, which in turn calls the appropriate abstract functions for the given abstract domain.

It is important to note that the only modification that was needed for extending PLAI to CLP languages was the elimination of a “unifiability” test performed before executing the abstract entry function [121, 122, 125]. This test is performed in the analysis of traditional LP languages in order to avoid analyzing clauses whose head does not (syntactically) unify with the current subgoal (e.g., the subgoal \(\text{append}([\text{x} s], \text{y} s, \text{z})\) and the head \(\text{append}([\text{ }, \text{y} s, \text{y} s])\). Naturally, the domain-dependent abstract functions had to be implemented and incorporated into the system but almost all the existing implementation was reused. We believe that this strongly supports our claim regarding the practical usefulness of the approach, specially considering that the resulting system can analyze reasonably sized programs in quite reasonable times.
6.5.1 Benchmark Programs

A relatively wide range of programs has been used as benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sumlist</td>
<td>sum of a list of numbers</td>
</tr>
<tr>
<td>listlength</td>
<td>length of a list</td>
</tr>
<tr>
<td>mortgage</td>
<td>well-known mortgage</td>
</tr>
<tr>
<td>laplace</td>
<td>Dirichlet problem for Laplace’s equation</td>
</tr>
<tr>
<td>norm_laplace</td>
<td>Normalized laplace</td>
</tr>
<tr>
<td>runge-kutta</td>
<td>differential equation solver (runge-kutta method)</td>
</tr>
<tr>
<td>trapezoid</td>
<td>differential equation solver (trapezoid method)</td>
</tr>
<tr>
<td>sendmm</td>
<td>send+more=money puzzle</td>
</tr>
<tr>
<td>rectangle</td>
<td>fill a rectangle with squares</td>
</tr>
<tr>
<td>meal</td>
<td>meal program</td>
</tr>
<tr>
<td>vector_matrix</td>
<td>vector-matrix addition and multiplication</td>
</tr>
<tr>
<td>dnf</td>
<td>propositional formula in disjunctive normal form</td>
</tr>
<tr>
<td>power</td>
<td>minimizes the production cost of power stations</td>
</tr>
<tr>
<td>mining</td>
<td>optimizes the revenue of a mine</td>
</tr>
<tr>
<td>num</td>
<td>number to letters-phonems translation</td>
</tr>
</tbody>
</table>

Table 6.1: Description of the benchmark programs

While Table 6.1 describes their function, Table 6.2 gives some data (similar to that provided for the benchmarks used in the previous chapter) which, in our view, provide good insight into the complexity of each benchmark, useful for the interpretation of the results:

- AgV, MV are respectively the average and maximum number of variables in each clause analyzed (note that dead code is not considered);

- Ps is the total number of predicates analyzed;

- Non, Rec are respectively the percentage of non-recursive and recursive predicates;

- Gs is the total number of different goals solved in analyzing the program, i.e., the total number of syntactically different calls.

151
As mentioned in the previous chapter, the number of variables in a clause affects the complexity of the analysis because the abstract functions greatly depend on the number of variables involved. Note that when abstract unification is performed, the variables of both the subgoal and the head of the clause to be unified have to be considered. Therefore, the number of variables involved in an abstract unification can be greater than the maximum number of variables shown in the table. For example, in \texttt{laplace} there is a recursive call involving 21 non definite variables. The number of recursive predicates affects the complexity of the fixpoint algorithm possibly increasing the number of iterations needed. Finally, the number of different goals gives an idea of the number of entries in the memoization table, possibly augmented if different call patterns for the same goal are found.

### 6.5.2 Results and Discussion

The experiments aim at evaluating the efficiency an accuracy of the analyzer. We have analyzed each benchmark\(^9\) ten times and then computed the average. The experiments have been done in a SparcStation 10, with one processor, using SICStus 2.1 in native

\(^9\)Some benchmarks have several usual call patterns. In those cases we have analyzed them for (the abstraction of) each one of such call patterns.
The analysis times show that the analysis can handle reasonable programs, even more if one takes into account that the analyzer has been implemented as a prototype, and therefore can be greatly optimized, and the high level of recursiveness of the programs. The less satisfactory results from the point of view of efficiency are those given for the laplace program. This is due to the recursive call mentioned above in which 21 non definite variables are involved when performing the abstract entry and abstract exit procedures. When this recursive call is normalized (as performed in the norm_laplace program) the time becomes reasonable again.

Regarding the accuracy of the information inferred, we have measured for each benchmark the total number of variables known to be definite by the analyzer and the total number of variables for which no definite information is inferred. This information has been compared with the abstraction of the concrete constraints obtained at each program point in real executions. Note that since the analyzer gives information at all program points, the number of variables for which information is inferred results from multiplying, for each clause, the number of variables by the number of points in the
Table 6.4 presents the accuracy results for each benchmark. The meaning of each column is as follows:

- **Analyzer**: accuracy of the information inferred by the analyzer. It shows for each benchmark the percentage of variables known to be definite (Def.) and the percentage of variables for which no definite information is inferred (Top).

- **Real Exec**: accuracy of the information obtained by analyzing the concrete constraints obtained at each program point in a real execution. The information shown is the same as above.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Analyzer</th>
<th>Real Exec.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Def.</td>
<td>Top</td>
</tr>
<tr>
<td>listlength</td>
<td>66.7</td>
<td>16.7</td>
</tr>
<tr>
<td>sumlist</td>
<td>66.7</td>
<td>16.7</td>
</tr>
<tr>
<td>num</td>
<td>30.4</td>
<td>47.6</td>
</tr>
<tr>
<td>meal</td>
<td>28.6</td>
<td>71.4</td>
</tr>
<tr>
<td>laplace</td>
<td>0.0</td>
<td>89.6</td>
</tr>
<tr>
<td>norm_lap</td>
<td>0.0</td>
<td>69.4</td>
</tr>
<tr>
<td>rectangle</td>
<td>0.0</td>
<td>81.8</td>
</tr>
<tr>
<td>mortgage</td>
<td>5.6</td>
<td>88.9</td>
</tr>
<tr>
<td></td>
<td>77.8</td>
<td>22.2</td>
</tr>
<tr>
<td>runge_kutta</td>
<td>63.1</td>
<td>34.0</td>
</tr>
<tr>
<td>trapezoid</td>
<td>14.6</td>
<td>46.6</td>
</tr>
<tr>
<td>dnf</td>
<td>76.8</td>
<td>23.2</td>
</tr>
<tr>
<td>sendmm</td>
<td>11.2</td>
<td>55.2</td>
</tr>
<tr>
<td>mining</td>
<td>19.8</td>
<td>55.8</td>
</tr>
<tr>
<td>power</td>
<td>18.5</td>
<td>53.5</td>
</tr>
<tr>
<td>vector_matrix</td>
<td>78.4</td>
<td>18.4</td>
</tr>
<tr>
<td></td>
<td>43.1</td>
<td>41.6</td>
</tr>
</tbody>
</table>

Table 6.4: Accuracy Results

It is important to note that the results for the concrete information given for mining and power do not correspond to the whole program, i.e., there are a few clauses for which no information is given. This is because the concrete execution did not reach such clauses and the high complexity of those benchmarks makes the task of inferring the
information by hand very difficult. However, as we will see, the information obtained is still useful for establishing the appropriate comparisons.

Also, note that the percentage of variables for which definite dependencies have been inferred can be obtained by simply subtracting from 100 the percentage of variables which are definite and for which no information is inferred. On the other hand, measuring the accuracy of such dependencies in a way that can be easily shown in a table is difficult. Thus, although we will not show such information, we will summarize the accuracy of the information when discussing the results for each benchmark.

The results of table 6.4 show that the analyzer accurately infers the definiteness of the variables for the following programs listlength, sumlist, num, meal, laplace, dnf, sendmm, and vector matrix when it is called with the first call pattern. Furthermore, in all these cases, except for laplace and sendmm, the dependencies are also accurately inferred. The reason is that definiteness is propagated in the usual way, without using the full power of the constraint solvers. The case of num is interesting since although it uses the properties of the PrologIII constraint solver for lists to concatenate lists, definiteness is still propagated by means of the definiteness dependencies, without, for example, using the length of a list to uniquely define it. In the case of laplace and sendmm, although the modes (i.e., definite and top) are accurately inferred, the dependencies are not since they are based on the use of the simplex algorithm. The case of sendmm is relevant, since definiteness is inferred thanks to the use of builtins which uniquely define the variables.

For the rest of the programs, although the accuracy of the information inferred is reasonable, it is not optimal. In general, and as would be expected, the analyzer accurately infers definiteness information when definiteness is explicit or can be propagated, but is (safely) inaccurate when definiteness is the result of solving a system of equations, or when nonlinear constraints, inequations, length constraints, etc., appear. Arguably, the results are quite acceptable for an analysis which is not specialized for any particular constraint system and is simple to implement.

\section{Chapter Conclusions}

We have presented a practical approach to the data-flow analysis of programs written in constraint logic programming (CLP) languages using abstract interpretation. We have shown that, from the framework point of view, it suffices with quite simple extensions of traditional analysis methods, with the advantage that such methods have already been proved useful and practical and that efficient fixpoint algorithms have already been developed for them. Along this line we have proposed a simple but quite general
extension to the analysis of CLP programs of Bruynooghe's traditional framework in which constraints are viewed not as "suspended goals" but rather as new information in the store, following the traditional view of CLP. Using this approach, a complete, constraint system independent, abstract analysis has been presented for approximating definiteness information which is of quite general applicability since it uses in its implementation only constraints over the Herbrand domain. We have also presented some results from an implementation of this analysis which show that the approach is indeed practical.
Chapter 7

Analyzing Logic Programs with Dynamic Scheduling

As mentioned in Chapter 4, dynamic scheduling overcomes the problems associated with traditional languages and their fixed scheduling, but it has a significant cost. Furthermore, since not only constraints but also atoms can be dynamically scheduled, global data-flow analyses used in the compilation of traditional (C)LP languages, such as mode analysis, are not correct with dynamic scheduling. This means that compilers, and in particular parallelizing compilers, for languages with dynamic scheduling are currently unable to perform optimizations which improve execution speed of traditional languages by an order of magnitude [71, 82, 146, 133, 155]. However, it is not simple to extend analyses for traditional Prologs to languages with dynamic scheduling, as in existing analyses the fixed scheduling is crucial to ensure correctness and termination.

Here we develop a framework for global data-flow analysis of logic languages with dynamic scheduling. This provides the basis for optimizations which remove the overhead of dynamic scheduling and promises to make the performance of logic languages with dynamic scheduling competitive with traditional (C)LP languages.

First, we give a denotational semantics for languages with dynamic scheduling. This provides the semantic basis for our generic analysis. The main difference with denotational definitions for traditional languages is that sequences of delayed literals must also be abstracted and are included in “calls” and “answers”. A key feature of the semantics is to approximate sequences of delayed literals by multi-sets of literals which are annotated to indicate if they are possibly delayed or if they are definitely delayed. The use of multi-sets instead of sequences greatly simplifies the semantics with, we believe, little loss of precision. This is because in most “real” programs delayed literals which wake at the same time are independent while delayed literals which are dependent
will be woken at different times.

Second, we give a generic global data-flow analysis algorithm which is based on the denotational semantics. Correctness is formalized in terms of abstract interpretation [38]. The analysis gives information about call arguments and the delayed calls, as well as implicit information about possible call schedulings at runtime. The analysis is generic in the sense that it has a parametric domain and various parametric functions. The parametric domain is the description chosen to approximate sets of term equations. Different choices of descriptions and associated parametric functions provide different information and give different accuracy. The parametric functions also allow the analysis to be tailored to particular system or language dependent criteria for delaying and waking calls. Implementation of the analysis is by means of a "memoization table" in which information about the "calls" and their "answers" encountered in the derivations from a particular goal are iteratively computed.

Finally, we demonstrate the utility and practical importance of the data-flow analysis algorithm. We sketch an example instantiation of the generic analysis which gives information about groundness and freeness of variables in the delayed and actual calls. Information from the example analysis can be used to optimize target code in many different ways. In particular, it can be used to reduce the overhead of dynamic scheduling by removing unnecessary tests for delaying and awakening and by reordering goals so that literals are not delayed. It can also be used to perform optimizations used in the compilation of traditional Prolog such as: recognizing determinate code and so allowing unnecessary backtrack points to be deleted; improving the code generated for unification; recognizing calls which are "independent," based on the definitions provided in Chapter 4, and so allow the program to be run in parallel, etc. Preliminary test results, given here, show that the analysis and associated optimizations used to reduce the overhead of dynamic scheduling give significant improvements in the performance of these languages.

To our knowledge this is the first time a full global data-flow analysis of logic programming languages with delay is considered. Related work includes Marriott et. al. [109] which gives a data-flow analysis for a logic programming language in which negated calls are delayed until their arguments are fully ground. However the analysis does not generalize to the case considered here as correctness relies on calls only being woken when all of their arguments are ground. Other related work is the global analysis of concurrent constraint programming languages [28, 29, 31, 104]. These languages differ from the languages considered here as they do not have a default left-to-right scheduling but instead the compiler or interpreter is free to choose any scheduling. Thus, program analysis must be correct for all schedulings. In our setting, knowledge
of the default scheduling potentially allows much more precise analysis. Related work also includes Gudeman et al. [61] and Debray [51] which investigate local analyses and optimization for the compilation of Janus, a concurrent constraint programming language. The optimizations include reordering and removal of redundant suspension conditions. Debray [48] studies global analysis for compile-time fixed scheduling rules other than left-to-right. However this approach does not work for dynamic scheduling, nor for analyses to determine “freeness” information. Finally, Hams [66] gives an analysis for improving the resitution mechanism in functional logic programming languages. This analysis handles the delay and waking of constraints, but does not easily extend to handle atoms as these may spawn subcomputations which in turn have delayed atoms.

In Section 7.1 we give the operational semantics of logic languages with dynamic scheduling. In Section 7.2 we review abstract interpretation and introduce various descriptions used in the analysis. In Section 7.3 we give the denotational semantics. In Section 7.4 we give the generic analysis, and in Section 7.5 we give modifications which ensure termination. Section 7.6 presents some performance results.

### 7.1 Operational Semantics

The operational semantics that we start from is that defined in Section 4.2, modified in Section 4.5. Finally, in order not to unnecessarily complicate the definition of the denotational semantics, we will restrict ourselves to the case in which only atoms are allowed to be dynamically scheduled. Extending the framework for dealing with this case is both intuitively and formally straightforward, but results in a very complex and non-intuitive definition.

Therefore, a state \( \langle G, c, D \rangle \) consists of the current literal sequence or “goal” \( G \), the current constraint \( c \), and the current sequence of delayed atoms \( D \). Literals in the goals are processed left-to-right. If the literal is an equality constraint, and it is consistent with the current equality constraints, it is added to these. Delayed atoms woken by the addition are processed. If the literal is an atom, it is tested to see if it is delayed. If so it is placed in the delayed atom sequence, otherwise it is replaced by the body of a clause in its definition.

The transitions in the transition system are:

- \( \langle a : G, c, D \rangle \rightarrow_d \langle G, c, a : D \rangle \) if \( a \) is an atom and \( \text{delay}(a, c) \) holds.

- \( \langle a : G, c, D \rangle \rightarrow_r \langle B :: G, c \land (a = h), D \rangle \) if \( a \) is an atom, \( r \equiv h \leftarrow B \) is a clause of program \( P \) renamed to new variables selected by the search rule, and \( h \) and \( a \) have the same predicate symbol.
\[ \langle a : G, c, D \rangle \rightarrow_r \text{fail} \text{ if } a \text{ is an atom and, for every clause } r \equiv h \leftarrow B \text{ of } P, \text{ } h \text{ and } a \text{ have different predicate symbols.} \]

\[ \langle c' : G, c, D \rangle \rightarrow_{cw} \langle D' :: G, c \land c', D \setminus D' \rangle \text{ if } c' \in \text{Cons, } \text{consistent}(c \land c') \text{ holds and } D' = \text{woken}(D, c \land c'). \]

\[ \langle c' : G, c, D \rangle \rightarrow_{cw} \text{fail} \text{ if } c' \text{ is a constraint and } \text{consistent}(c \land c') \text{ does not hold.} \]

We will assume all the properties established in previous chapters related to the \emph{consistent} function, namely that it does not take variable names into account and, if a constraint is said to be consistent, all constraints entailed are also consistent. Analogously, the parametric functions \emph{delay} and \emph{woken} satisfy that there is a congruence between the conditions for delaying a call and waking it, and \emph{delay} behaves reasonably: it should not take variable names into account, it should only be concerned with the effect of the constraint on the variables in the atom, and if an atom is not delayed, adding more constraints should never cause it to delay.

The problems related to the operational semantics of dynamically delayed languages is that real implementations are also “dynamic” in the sense that there are many implementation dependent decisions which can significantly affect the semantics of the program. For example, the order of the calls returned by \emph{woken} is system dependent. Furthermore, although we have assumed that the \( \rightarrow_{cw} \) transition rule of the general operational semantics is always applied after conjuncting a constraint \( c' \) with the store, many real implementations do not behave in such way, allowing such transition to be applied only either before the \( \rightarrow_r \) transition rule or when the last literal is a constraint. Since it is straightforward to modify both the operational and denotational semantics to model this behavior, we will not discuss it in detail. A perhaps more subtle issue is that related with the equivalence between a non normalized program and its associated normalized program, when the treatment of all constraints resulting from the the equation between the goal and the head of a rule is considered as an “atomic” step, i.e., no atom is allowed to be woken before all constraints have been processed. In this case, considering the semantics given above, a non normalized program would not be equivalent to its associated normalized program. This problem can also be easily solved by including a special symbol after the sequence of constraints resulting form the head normalization, and not allowing any atom to be woken before processing such symbol.
7.2 Abstract Interpretation

As pointed out before in the thesis, in abstract interpretation an analysis is formalized as a non-standard interpretation of the data types and functions over those types. Correctness of the analysis with respect to the standard interpretation is argued by providing an approximation relation which holds whenever an element in a non-standard domain describes an element in the corresponding standard domain.

In the analysis of dynamically scheduling languages we will need to describe sequences of delayed atoms and sequences of woken atoms. Because of the inherent imprecision in analyzing programs with a dynamic computation rule, we cannot always be definite that a particular atom is in the sequence, but may only know that it is possibly in the sequence. Further, it is difficult to keep track of all possible sequence orderings. Hence, we will describe atom sequences by a multi-set of annotated atoms in which each atom is annotated with def if it definitely appears in the sequence and pos if it possibly appears in the sequence. For example, the multi-set \{\langle p(x), \text{pos} \rangle, \langle q(y), \text{def} \rangle \} describes only the sequences p(x) : q(y) : nil, q(y) : p(x) : nil and q(y) : nil. More formally, let \( \text{Var} \) be the set of variables, \( \text{Atom} \) the set of atoms, \( \text{Cons} \) the set of constraints, \( \text{Lit} \) the set of literals, and \( \text{Prog} \) the set of programs. Then:

\[
\begin{align*}
\text{Ann} & = \{\text{def}, \text{pos}\} \\
\text{AnnAtom} & = \text{Atom} \times \text{Ann} \\
\text{AnnMSet} & = \varnothing(\text{AnnAtom})
\end{align*}
\]

Let \( D^* \in \text{AnnMSet} \). Define \( \text{def}(D^*) \) to be the multi-set of atoms in \( D^* \) that are annotated with \( \text{def} \) and \( \text{all}(D^*) \) to be the multi-set of all atoms in \( D^* \), that is the atoms annotated with either \( \text{pos} \) or \( \text{def} \). The atom sequence description is

\[
\langle \text{AnnMSet}, \alpha^{\text{AnnMSet}}, \text{Atoms} \rangle
\]

where \( \alpha^{\text{AnnMSet}} \) is defined by

\[
\alpha^{\text{AnnMSet}}(D) = \{ \langle l, \text{def} \rangle \mid l \text{ in } D \}
\]

and \( \text{AnnMSet} \) is ordered by \( D_1^* \preceq D_2^* \) iff

\[
\text{all}(D_1^*) \subseteq \text{all}(D_2^*) \land \text{def}(D_2^*) \subseteq \text{def}(D_1^*).
\]

It follows that the annotated multi-set \( D^* \) approximates the sequence \( D \) iff all atoms in \( D \) are in \( \text{all}(D^*) \) and every atom in \( \text{def}(D^*) \) is in \( D \).

We will also be interested in describing constraints. The analysis given in the next section is generic in the choice of description. We require that the description chosen,
\(\langle \text{ACons}, \alpha^{\text{ACons}}, \text{Cons} \rangle\) say, satisfies

\[
\alpha^{\text{ACons}}(c) = \bot_{\text{ACons}} \iff c \rightarrow \text{false}
\]

where \(\bot_{\text{ACons}}\) is the least element in \(\text{ACons}\).

One example of a constraint description is the standard constraint description,

\[
\langle \text{Cons}^+, \lambda c : c, \text{Cons} \rangle
\]

in which constraints describe themselves. More precisely, the description domain is \(\text{Cons}^+\) which is the set of constraint with a new top element \(\top\). \(\text{Cons}^+\) is a complete lattice ordered by

\[
c \leq c' \iff c \rightarrow \text{false} \text{ or } c \leftrightarrow c' \text{ or } c' \equiv \top.
\]

The abstraction function is the identity function, as the best description of a constraint is just itself.

### 7.3 Denotational Semantics

In this section we give a generic denotational semantics for programs with dynamic scheduling. Correctness of the denotational semantics depends on the following results about the operational semantics. The first proposition means that we can find the answers of a state in terms of its constituent atoms; the second means that we can consider states modulo variable renaming; the third that we can restrict the constraint to the variables in the delayed atoms and the goal.

**Proposition 7.3.1** Let \(P \in \text{Prog}\), \(a \in \text{Atom}\) and \(\langle a : G, c, D \rangle \in \text{State}\). Then \(\text{answer}_P(\langle a : G, c, D \rangle)\) is

\[
\bigcup \{ \text{answer}_P(\langle G, c', D' \rangle) \mid \langle c', D' \rangle \in \text{answer}_P(\langle a, c, D \rangle) \}.
\]

**Proposition 7.3.2** Let \(P \in \text{Prog}\), \(\rho \in \text{Ren}\), and \(S \in \text{State}\).

\[
Q \in \text{answer}_P(S) \iff \rho(Q) \in \text{answer}_P(\rho(S)).
\]

**Proposition 7.3.3** Let \(P \in \text{Prog}\) and \(\langle G, c, D \rangle \in \text{State}\).

\[
\text{answer}_P(\langle G, c, D \rangle) = \bigcup \{ \langle c' \land c, D' \rangle \mid \langle c', D' \rangle \in \text{answer}_P(S) \},
\]

where \(S\) is \(\langle G, \text{exists}_{-\text{vars}}(G, D) \rangle, D\).
Taken together these propositions mean that we can find the answers to a state as long as we know the answers to the “canonical” calls encountered when processing the state where a canonical call is a call that represents all of its variants and in which the constraint is restricted to the variables of the call atom and the delayed literals. This is the basic idea behind the denotational semantics as the denotation of a program is simply a mapping from calls to answers.

The last proposition means that the meaning of a goal is independent of the order that the atoms are scheduled. Thus we can ignore the sequencing information associated with delayed atoms and treat them as multi-sets. It is variant of Theorem 4 in Yelick and Zachary [158].

**Proposition 7.3.4** Let $P$ be a program and $\langle G, \theta, D \rangle$ be a state. If $G'$ is a rearrangement of $G$ then,

$$answer_P(\langle G, \theta, D \rangle) = answer_P(\langle G', \theta, D \rangle).$$

In the denotational semantics literals, bodies, clauses and programs are formalized as “environment” transformers where an environment consists of the current constraint description and an annotated multi-set of delayed atoms. In a sense an environment is the current “answer”. Thus an environment has type $Env = ACons \times AnnMSet$ and the denotation of a program has type $Den = Atom \to Env \to \wp(Env)$ as it maps a call to its set of answers.

The denotational semantics has the following semantic functions:

- $P : Prog \to Den$
- $Q : Prog \to Den \to Den$
- $B : Lit^* \to \wp(Var) \to Den \to Env \to \wp(Env)$
- $L : Lit \to \wp(Var) \to Den \to Env \to \wp(Env)$
- $A : Atom \to \wp(Var) \to Den \to Env \to \wp(Env)$
- $C : Cons \to \wp(Var) \to Den \to Env \to \wp(Env)$

the following auxiliary functions:

- $lookup : Atom \to \wp(Var) \to Den \to Env \to \wp(Env)$
- $wmset : AnnMSet \to \wp(Var) \to Den \to Env \to \wp(Env)$
- $watom : AnnAtom \to \wp(Var) \to Den \to Env \to \wp(Env)$

and the following parametric functions:

- $Awake : Atom \to ACons \to ACons$
\[\text{Adelay} : \text{Atom} \rightarrow \text{ACons} \rightarrow \text{ACons}\]
\[\text{Acomb} : \text{ACons} \rightarrow \text{ACons} \rightarrow \text{ACons}\]
\[\text{Arestrict} : \varphi(\text{Vars}) \rightarrow \text{ACons} \rightarrow \text{ACons}\]
\[\text{Aadd} : \text{Cons} \rightarrow \text{ACons} \rightarrow \text{ACons}\]
\[\text{Awoken} : \text{AnnMSet} \rightarrow \text{Cons} \rightarrow \text{ACons} \rightarrow \text{AnnMSet}\]
\[\text{Adelayed} : \text{AnnMSet} \rightarrow \text{Cons} \rightarrow \text{ACons} \rightarrow \text{AnnMSet}.\]

Let \(\text{defn_P}(a, W)\) denote the set of variants of clauses in a program \(P\) such that each variant has the atom \(a\) as a head and has variables disjoint from the set of variables \(W \setminus \text{vars}(a)\). The semantic and auxiliary functions are defined by:

\[
P \; [P] \; a \; e \quad = \quad A \; [a] \; \text{vars}(e) \; \text{lfp}(Q \; [P]) \; e
\]
\[
Q \; [P] \; d \; a \; e \quad = \quad \text{let } V = \text{vars}(e) \text{in} \quad \bigcup \{ (B \; [B] \; V \cup \text{vars}(a \leftarrow B) \; d \; e) \mid (a \leftarrow B) \in \text{defn_P}(a, V) \}
\]
\[
B \; [\emptyset] \; W \; d \; e \quad = \quad \{ e \}
\]
\[
B \; [B] \; W \; d \; e \quad = \quad \bigcup \{ (B \; [B] \; W \; d \; e') \mid e' \in (L \; [\emptyset] \; W \; d \; e) \}
\]
\[
L \; [\emptyset] \quad = \quad \text{if } i \in \text{Atom then } (A \; [i]) \text{ else } (C \; [\emptyset])
\]
\[
A \; [a] \; W \; d \; \langle AC, D^* \rangle \quad = \quad \text{if } \text{Awake}(a, AC) = \bot_{\text{ACons}}, \text{then } \{ \langle AC, \langle a, \text{def} \rangle \cup D^* \rangle \}
\quad \text{else if } \text{Adelay}(a, AC) = \bot_{\text{ACons}}, \text{then } (\text{lookup } a \; W \; d \; \langle AC, D^* \rangle)
\quad \text{else } \{ \langle \text{Adelay}(a, AC), \langle a, \text{pos} \rangle \cup D^* \rangle \}
\quad \bigcup \ (\text{lookup } a \; W \; d \; \langle \text{Awake}(a, AC), D^* \rangle)
\]
\[
C \; [\emptyset] \; W \; d \; \langle AC, D^* \rangle \quad = \quad \text{let } AC' = \text{Aadd}(c, AC) \text{ in} 
\quad \text{if } AC' = \bot_{\text{AEqns}}, \text{then } \emptyset; \text{ else} 
\quad (\text{wmset } \text{Awoken}(D^*, c, AC) \; W \; d \; \langle AC', \text{Adelayed}(D^*, c, AC) \rangle)
\]
\[
\text{lookup } a \; W \; d \; \langle AC, D^* \rangle \quad = \quad \text{let } V = \text{vars}(a) \cup \text{vars}(D^*) \text{ in} 
\quad \text{let } E = d \; a \; \langle \text{Arestrict}(V, AC), D^* \rangle \text{ in} 
\quad \{ (\text{Acomb}(AC, AC'), D^*) \mid (AC', D^*) \in (\text{rename } E \; V \; W) \}
\]
\[
\text{wmset } \emptyset \; W \; d \; e \quad = \quad \{ e \}
\]
\[
\text{wmset } (a^* \cup D^*) \; W \; d \; e = \bigcup \{ (\text{wmset } D^* \; W \; d \; e') \mid e' \in (\text{watom } a^* \; W \; d \; e) \}
\]
\[
\text{watom } (a, \text{def}) \; W \; d \; \langle AC, D^* \rangle \quad = \quad \text{lookup } a \; W \; d \; \langle AC, D^* \rangle
\]
\[
\text{watom } (a, \text{pos}) \; W \; d \; \langle AC, D^* \rangle \quad = \quad \{ \langle AC, D^* \rangle \} \cup (\text{lookup } a \; W \; d \; \langle \text{Awake}(a, AC), D^* \rangle)
\]

The semantics is generic as it is parametric in \(\text{ACons}\) the constraint descriptions and various parametric functions. The semantic functions associated with programs \(P\), queries \(Q\), clause bodies \(B\), and literals \(L\), need little explanation. The only points
to note is that the variable set \( W \) is passed around so as to ensure that there are no variable (re)namings.

The function \( A \) gives the meaning of an atom for the current denotation. Consider the call \( A \llbracket a \rrbracket W d \langle AC, D' \rangle \). There are three cases to consider: the first is when the atom \( a \) is delayed for all constraints approximated by \( AC \), the second is when \( a \) is not delayed for any constraints approximated by \( AC \), and the third is when \( a \) is delayed for some constraints approximated by \( AC \), but not all. \( A \) is defined in terms of the parametric functions \( Awake \) and \( Adelay \). The call \( Awake(a, AC) \) returns a description of those constraints which are described by \( AC \) and for which \( a \) will not delay. Conversely, \( Adelay(a, AC) \), returns a description of those constraints which are described by \( AC \) and for which \( a \) will delay. More exactly, \( Awake \) and \( Adelay \) should satisfy:

\[
\{ Awake(a, AC) \} \propto \{ c \mid AC \propto c \land \neg delay(a, c) \}
\]
\[
\{ Adelay(a, AC) \} \propto \{ c \mid AC \propto c \land delay(a, c) \}.
\]

Note that \( Awake(a, AC) = \bot_{AC_{cons}} \) implies

\[
AC \propto c \Rightarrow delay(a, c),
\]

and \( Adelay(a, AC) = \bot_{AC_{cons}} \) implies

\[
AC \propto c \Rightarrow \neg delay(a, c).
\]

The auxiliary function \( lookup \) is used to find the denotation of an atom which possibly does not delay. The call \( lookup a W d \langle AC, D' \rangle \), returns the denotation according to \( d \) of \( a \) with environment \( \langle AC, D' \rangle \). However there are complications because \( d \) only handles “canonical calls”. Hence \( lookup \) must (1) restrict \( AC \) to the variables in the call; (2) rename the variables introduced in the delayed atoms in the answers so that they do not interfere with the variables in \( W \); and (3), combine the constraint description with that of the original call so as to undo the result of the restriction. \( Lookup \) is defined in terms of the parametric functions \( Acomb \) and \( Arestrict \). \( Acomb \) combines two constraint descriptions and should approximate the function \( add \), defined by

\[
add(c, c') = c \land c'.
\]

\( Arestrict \) restricts a constraint description to a set of variables and should approximate the function \( restrict \) defined by

\[
restrict W \theta = \exists_W \theta.
\]
The definition also makes use of the function call \textit{rename E V W} which returns a variant of the environments \(E\) which is disjoint from the variables \(W\) but which leaves the variables in \(V\) unchanged. More exactly it returns \(\rho(E)\) where \(\rho\) is a renaming such that for all \(v \in V\), \(\rho(v) = v\) and \(\text{vars}(\rho(E)) \cap (W \setminus V) = \emptyset\).

Constraints are handled by the semantic function \(C\). The function call, \(C \llbracket e \rrbracket W d \langle AC, D' \rangle\), first adds the constraint \(c\) to \(AC\) and tests for satisfiability. If this succeeds, it then wakes up the atoms in \(D'\), and processes these. The definition is parametric in the functions \textit{Aadd}, \textit{Awoken} and \textit{Adelayed}. The function \textit{Aadd} adds a constraint to an constraint description and must approximate the function \textit{add} defined previously. \textit{Awoken} returns the multi-set of atoms that will be possibly and definitely woken by adding a constraint to an constraint description and \textit{Adelayed} returns the multi-set of atoms that will possibly and definitely remain delayed. \textit{Awoken} must approximate \textit{diffwoken} and \textit{Adelayed} must approximate \textit{diffdelay} where these are defined by

\[
diffwoken(D, c, c') = \text{woken}(D \setminus \{\text{woken}(D, c')\}, c \land c')
\]
\[
diffdelay(D, c, c') = D \setminus \text{woken}(D, c \land c').
\]

Note that \textit{Adelayed} may change the annotation of a delayed atom from \textit{def} to \textit{pos} and that \textit{Awoken} returns a multi-set of woken atoms which are also annotated.

The woken atoms are handled by the auxiliary functions \textit{wmsset} and \textit{watom} almost exactly as if they were a clause body, the only difference is to handle the \textit{pos} annotated atoms.

The \textit{standard} denotational semantics, \(P_{\text{std}}\), is obtained by from the denotational semantics by instantiating \textit{ACons} to the standard constraint descriptions and instantiating the parametric functions to the function they are required to approximate, for instance \textit{Aadd} and \textit{Acomb} are both instantiated to \textit{add}. Using the four propositions given at the start of this section, it is possible to show that the denotational semantics is correct:

\textbf{Theorem 7.3.5} Let \(D \in \text{Atoms}\), \(c \in \text{Cons}\), \(a \in \text{Atom}\), and \(P \in \text{Prog}\). Then

\[
P_{\text{std}} [P] a \langle c, D' \rangle = \text{answer}_P((a: \text{nil}, c, D))
\]

where \(D' = \alpha^{\text{AnnMSet}}(D)\).

Using results from abstract interpretation theory it follows that analyses based on the semantics are correct:

\textbf{Theorem 7.3.6} Let \(e \in \text{Env}\), \(a \in \text{Atom}\), \(P \in \text{Prog}\). If \(e \preceq \langle c, D \rangle\),

\[
(P [P] a e) \preceq \text{answer}_P((a: \text{nil}, c, D)).
\]
Actually the denotational semantics does not exactly give the information a compiler requires for the generation of efficient code. This is because we are primarily interested in removing unnecessary tests for delaying and improving the code for unification. Therefore, we must obtain information about the call patterns. That is, for each atom $A$ appearing in the program we want to know whether the calls to the atom initially delay, and when each call to $A$ is eventually reduced, perhaps after being delayed, the value of the current equation restricted to the variables in $A$. It is straightforward to modify the denotational semantics to collect this information for atoms which are not delayed. For the case of atoms which are delayed it is more difficult as although treating the delayed atoms as a multi-set does not affect the answers, if more than one atom is woken it may affect the calls made in the evaluation. Since the solutions to be taken significantly depend on the particular implementation considered, we will ignore this extra complication but note that it has been dealt with in the analyzer used to obtain the results presented in Section 7.6.

### 7.4 Implementation

The denotational equations given in the previous section can be considered as a definition of a class of program analyses. Read naively, the equations specify a highly redundant way of computing certain mathematical objects. On the other hand, the denotational definitions can be given a "call-by-need" reading which guarantees that the same partial result is not repeatedly recomputed and only computed if it is needed for the final result. With such a call-by-need reading the definition of $P$ is, modulo syntactic rewriting, a working implementation of a generic data-flow analyzer written in a functional programming language.

In programming languages which do not support a call-by-need semantics, implementation is somewhat harder. To avoid redundant computations, the result of invoking atom $a$ in the context of environment $e$ should be recorded. Such memoing can be implemented using function graphs. The function graph for a function $f$ is the set of pairs $\{(e \mapsto f(e)) \mid e \in \text{dom}(f)\}$ where $\text{dom}(f)$ denotes the domain for $f$. Computation of a function graph is done in a demand-driven fashion so that we only compute as much of it as is necessary in order to answer a given query. This corresponds to the "minimal function graph" semantics used by Jones and Mycroft [93]. However, matters are complicated by the fact that we are performing a fixpoint computation and we must iteratively compute the result by means of the function's Kleene sequence.

This idea leads to a generic algorithm for the memoization based analysis of programs with dynamic scheduling. The algorithm extends memoization based analysis
for traditional Prolog. The analysis starts from a “call” and incrementally builds a memoization table. This contains tuples of “calls” and their “answers” which are encountered in derivations from the initial call. Calls are tuples of the form \( \langle a, AC, D^e \rangle \) where \( a \) is an atom, \( D^e \) is a multi-set of annotated atoms describing the sequence of delayed atoms and \( AC \) is a constraint description restricted to the variables in \( a \) and \( D^e \). An answer to a call \( \langle a, AC, D^e \rangle \) is of the form \( \langle AC', D'^e \rangle \) where \( D'^e \) is a multi-set of annotated atoms describing the sequence of delayed atoms and \( AC' \) is a constraint description restricted to the variables in \( a \) and \( D'^e \). Our actual implementation has two improvements which reduce the size of the memoization table.

The first improvement, is when adding an answer to the answers of call, to remove “redundant” answers and merge similar answers together. Answers \( \langle AC_1, D_1^e \rangle \) and \( \langle AC_2, D_2^e \rangle \) are merged into the single answer \( \langle AC_1 \cup AC_2, D_1^e \rangle \) whenever \( D_2^e \preceq D_1^e \).

The second improvement is to only consider calls modulo variable renaming. Entries in the memoization table are “canonical” and really represent equivalence classes of calls and answers.

Another possible improvement which has not been implemented yet is based on the observation that delayed atoms which are independent of the calling atom and the rest of delayed atoms, can never be woken when the call is executed. Such atoms need not be considered in the call as they will occur in each answer. Note that this is related to the partitioning of the sequence of delayed literals in the concrete operational semantics given in Section 4.5.

### 7.5 Termination

Correctness of the denotational semantics, Theorem 7.3.6, is not quite enough as it only guarantees partial correctness of an analysis, and, of course, we would also like the analysis to terminate. Given that all calls to the parametric functions terminate, the analysis will terminate if there are a finite number of calls in the memoization table and each call has a finite number of answers. This is true if the following two requirements are met. The first is that for each finite set of variables \( W \) there are only a finite number of descriptions which describe some constraints \( \exists_{-W} c \). This is the usual requirement for the termination of memoization based analysis of standard Prolog. The second requirement is that there is a bound on the size of the annotated multi-sets in both the calls and the answers. In this section we sketch two modifications to the analysis which ensure that only multi-sets of a bounded size need be considered, albeit at some loss of accuracy. In some sense, this is a form of widening [40], however correctness does not depend on the semantics of the description domain but rather on
properties of the program semantics.

The first modification allows us to only consider calls with annotated multi-sets of a bounded size. Correctness depends on the following property of the operational semantics:

**Proposition 7.5.1** Let $P \in \text{Prog}$ and $(G, \theta, D) \in \text{State}$. If $D = D' \cup D''$, $\text{answer}_P \langle G, \theta, D \rangle = \text{answer}_P \langle G :: D', \theta, D'' \rangle$. □

This means in the analysis that *lookup* can be modified to (1) remove annotated atoms $D'$ from the multi-set of delayed atoms, if it is too large, (2) proceed as before, and then (3) process $D'$ using a variant of $B$ which handles annotated atoms.

The second modification allows us to only consider answers with annotated multi-sets of a bounded size. Now a delayed atom $a$ can, if it is woken, only add constraints affecting variables in $a$ and variables which are local to its subcomputation. Thus in the analysis, when we encounter an answer $\langle AC, D' \rangle$ in which the multi-set $D'$ is too large, we can replace it by the answer $\langle AC', \{\top, \text{pos}\} \rangle$ where $\{AC'\}$ approximates

\[
\{c \land \exists\text{vars}(D') c' \mid AC \propto c \land c' \in Cons\}
\]

and $\langle \top, \text{pos} \rangle$ is a special annotated “atom” which signifies that there are possibly delayed atoms of indeterminate name. Note that $\langle \top, \text{pos} \rangle$ can never be woken.

With these two modifications the analysis will terminate whenever the usual termination requirements for memoization based analysis of standard Prolog are met.

We can also use the idea behind the second modification to analyze modules. The problem is that when analyzing a module in isolation from the context in which it will be used we have no idea of the delayed atoms associated with calls to the module. However, the delayed atoms can only affect variables in the initial call. Thus by taking the downward closure of the initial call, we are assured to obtain correct information about the calling patterns regardless of the atoms delayed in the actual call.

Another approach to ensure termination would be to approximate the delayed multi-set of atoms by a “star abstraction” [28] in which variants of the same atom are collapsed on to a single “canonical” atom.

**7.6 Performance Results**

We conclude with an empirical evaluation of the accuracy and usefulness of an implementation in Prolog of the analyzer presented. Our first results show that information from the analysis can be used to eliminate redundant delay declarations, leading to a large performance improvement. The last test illustrates how the analysis can be
used to guide optimizations which are performed for traditional Prolog. In this case we show how implicit independent and-parallelism as detected by the analyzer, using the conditions described in Chapter 4, can be used to speed-up the benchmark.

The analyzer used in the experiments is the one based on the Sharing+Freeness abstract domain evaluated in Chapter 5, suitably extended for our framework. The benchmarks used for the evaluation were: permute, the permute program presented in Section 4.1; qsort, the classical quick sort program using append; app3 which concatenates three lists by performing two consecutive calls to append; nrev which naively reverses a list; and neg, an implementation of safe negation using suspension on the groundness of the negated goal (a simple test of membership in a list). All benchmarks have been implemented in a reversible way, so that they can be used forwards and backwards, through the use of suspension declarations. The reasons for selecting benchmarks only using Herbrand constraints are twofold. Firstly, the only parallel system available at our site is &-Prolog system [71] introduced in Chapter 5. Although this system is being extended to deal with several kinds of constraint systems, the current version only deals with Herbrand constraints. Secondly, the analyzer used for the evaluation is not prepared either for dealing with other kind of constraints.

In the first test, the optimizations to eliminate unnecessary delaying were performed in two steps. The first step was to eliminate and/or relax suspension declarations as indicated by the analysis. The second step was to reorder the clause bodies provided the analysis indicated that it reduced suspension. It is important to note that although the obtained orderings are already implicit in the results of the (first) analysis, in order to eliminate suspension conditions that are redundant after the reordering, a second pass of the analysis is sometimes needed. The tests where performed with Sicstus Prolog 2.1, which is an efficient implementation of Prolog with a rich set of suspension primitives.

In order to give an idea of the accuracy of the analyzer and to help in understanding the efficiency results, we point out that in all cases but for permute the information provided by the analyzer was optimal. In the case of permute one condition can be relaxed beyond those inferred by the analyzer. In particular, for all the examples in their “forward” execution mode the analyzer accurately infers that no goal suspends and therefore all suspension declarations can be eliminated. With respect to the backwards execution, in all cases but neg the suspension conditions are either relaxed or eliminated. This does not occur for neg since the analyzer accurately infers that the existing groundness suspension condition is still needed for correctness. Finally, with respect to the optimizations where reordering is allowed, all backward executions are reordered in such a way that no suspension conditions are needed. Thus, we can conclude that the
accuracy results for the analyzer are encouraging.

Table 7.1 lists execution times, expressed in seconds, for the original benchmarks and the optimized versions. Each column has the following meaning: **Name** – program name, **Query** – number of elements in the list given as query, **P** – execution time for the program written in standard Prolog, i.e. with no suspension declarations, **S** – execution time for the program written with suspension declarations, **SO** – execution time for program written with suspension declarations and optimized by removing suspension declarations as dictated by the analysis information, **S/SO** – ratio between the last two columns, **R** – execution time for the program optimized by reordering the clause bodies as dictated by the analysis information, and **R/S** – ratio between **R** and **S** columns. In the **P** column **In** stands for non-termination, and **Er** stands for a wrong result or an
execution error (the fact that these cases appear shows the superiority of the version of the program with suspension declarations). Two sets of data (corresponding to two lines in the table) are given for each program, the first one corresponding to “forwards” execution of the program, the second to the “backwards” execution.

Note that in some cases the number of elements given as queries for forward execution are different from those used for the backward execution of the same program. The reason is the amount of time required by each query due to the different behavior when running forwards (one solution) and backwards (multiple solutions).

The results are rather appealing as they show that the optimizations based on relaxing and eliminating suspension declarations using the information provided by the analyzer allow the use of the more general version of the program written with suspension declarations without a performance penalty when executing the program in the mode that runs in Prolog. Furthermore, the analysis and resultant optimization also improves execution speed even if some suspensions still need to be used during execution. The optimizations based on reordering give even more impressive results. This is mainly explained by the fact mentioned above that for all programs the reordering has achieved the elimination of all suspension declarations.

Finally, in the last test, we show how information from the analysis can be used to perform optimizations used in the compilation of traditional Prolog. As an example we consider automatic parallelization based on the independent and parallelism model for CLP languages with dynamic scheduling developed in Chapter 4. The only program in which this kind of parallelism exists for the given queries is `qsort`. In this case the parallelism can be automatically exploited using existing tools given the information obtained from the analysis. This is because the analysis determines that there is no goal suspension in the reordered program and so the tools and techniques described in [77, 123] are applicable. These techniques can also be extended to deal with cases in which goals are delayed by extending the notion of dependence, but that is beyond the scope of this paper. A significant reduction in computation time is obtained from parallelism at least for the forward query. This is illustrated in Table 7.2, which shows results from running the forward query with the optimized program under the &-Prolog system [71] introduced in Chapter 5. Times are in seconds.

### 7.7 Chapter Conclusions

We have given a framework for global data-flow analysis of logic languages with dynamic scheduling. The framework extends memoization based analyses for traditional logic programming languages with a fixed left-to-right scheduling. Information from
analyses based on the framework can be used to perform optimizations which remove the overhead of dynamic scheduling and also to perform optimizations used in the compilation of traditional Prolog.
Chapter 8

Conclusions and Future Work

We have generalized the concept of independence and the a priori conditions developed for LP to the more general context of CLP and dynamically scheduled languages. We argue that this generalization is not only interesting in itself, but also yields new insights into the nature of independence and its relationships to other concepts. We have shown how some such insights can be even projected back into the Herbrand case. Firstly, we have clarified the relationship between search space preservation and the notion of independence, showing that search space preservation is not only a sufficient but also a necessary condition for ensuring correctness and efficiency (measured in terms of number of transitions) in the and-parallel model. Secondly, we have shown that search space preservation is not even sufficient to ensure efficiency when such efficiency is defined in terms of total execution cost rather than in terms of number of transitions. To solve this problem we have introduced the concept of “solver independence” which characterizes the properties of the constraint solver behavior when changing the order in which constraints are added. Finally, we have shown that the notions of independence are interesting for many optimizations, such as reordering and intelligent backtracking, which are based on modifying the usual sequential execution of a constraint logic program in order to improve its efficiency.

Having defined the notions of independence needed and knowing that global data-flow analysis has proven to be a key technology in program parallelization, we have focused on studying global data-flow analysis for program parallelization and generalizing it to deal with the languages under consideration. We have performed an exhaustive evaluation of global data-flow analysis techniques in the automatic parallelization task. This study is the most complete to date and the first one which analyzes the performance of global analysis in parallelization in terms of speedups. The results obtained from this experimental evaluation show that the effectiveness of the parallelization significantly depends on the the availability of the information provided by
global data-flow analysis. This led us to conclude that such analysis should also be useful in the context of dynamically scheduled constraint logic languages. Regarding CLP languages, we have shown how some of the techniques which have been used with great success in LP, and for which efficient fixpoint algorithms have already been developed, can relatively easily be extended to the analysis of CLP programs. In particular, we have proposed and implemented an extension to CLP of Bruynooghe's analysis and the PLAI framework. Finally, we have also proposed analysis technology for constraint logic programs with dynamic scheduling by proposing a denotational semantics for them and an abstraction based on this semantics. This approach has been empirically studied by implementing a complete global analyzer for constraint logic programs with dynamic scheduling, and studying its performance in the optimization of such programs.

We believe that the applicability of the techniques and concepts proposed goes well beyond what has been discussed explicitly in the preceding chapters. We will devote the rest of this concluding chapter to discussing this issue, in the context of and-parallelism (Section 8.1), as well as pointing out other avenues for future work (Section 8.2).

8.1 And-parallelism – A Unifying View

The concepts and techniques related to independence, parallelism, and analysis developed throughout the thesis were often explained or applied in the context of a particular model of and-parallelism: goal-level independent and-parallelism, i.e. and-parallelism in which the tasks to be executed simultaneously are subtrees corresponding to goals in a resolvent, and the principle being used to decide whether to parallelize those goals or not is independence.

A number of other models of (and-)parallelism have been proposed – most of them were reviewed in Chapter 1 – which are at first sight quite different to the goal-level independent and-parallel model. However, we believe these differences are smaller than they appear at first sight and there are clearly identifiable common concepts among these approaches. Our objective is to identify these aspects in order to develop a more uniform view for and-parallelism. We believe that this view is instrumental in conceptualizing the wide applicability that we will claim for the techniques developed in the thesis. In the discussion, we follow mainly Hermenegildo [76].

We start by observing that traditional proposals for parallel or concurrent models, such as those presented in Chapter 1, are often “bundled packages”, in the sense that they offer a combined solution affecting a number of issues such as choice of computation rule, concurrency, exploitation of parallelism, etc. This is understandable since a practical model has to offer solutions for all the problems involved. However, this often
makes it difficult to identify and study the fundamental principles involved.

First, it is useful to make a strict distinction between parallelism issues and computation-rule related issues. To this end, we define parallelism as the simultaneous execution of a number of independent sequences of computation steps, taken from those which would have to be performed in any case as determined by the computation rules. We call each such sequence a thread of execution. Note that as soon as there is an actual (i.e., run-time) dependency between two sequences, one has to wait for the other and therefore parallelism does not occur for some time. Thus, execution sequences (such as, for example, a sequence of resolutions) generally contain several threads. Exploiting parallelism means taking a fixed-size computation (determined by the computation rules), splitting the computation into independent threads related by dependencies (building a dependency graph), and assigning these threads to different agents.

In order to visualize this distinction, let us consider a typical or-parallel system. The performance gain obtained when computing all solutions for a goal with respect to that obtained by a sequential system has no relation with the computation rule. However, assume that only (any) one solution is needed and the shallower solution appears in branch different from that explored by the sequential system due to its associated computation rule. Then the or-parallel system may find this solution first, the possible performance gain coming in a fundamental way from a change in the computation rule, rather than from parallel execution itself.

A similar phenomenon appears in independent and-parallel systems if they incorporate a certain amount of “intelligent failure”, i.e. if whenever a parallel process cannot find a solution for a goal, it is capable of killing the other processes running in parallel. In such cases computation may be saved due to a smarter computation rule that is taking advantage of the knowledge of the independence of some goals rather than having really anything to do with the parallelism.

Another interesting example to consider is the Andorra-I system, which is based on the basic Andorra principle, as mentioned in Chapter 1. This principle can be seen as actually comprising two principles. One, related to the computation rule, says that deterministic reductions are executed first. This is potentially very useful in practice since it can result in a reduction of the number of resolutions needed to find a solution (although it can also lead to more steps or even a non terminating computation with respect to, for example, a left-to-right execution for a given program). The other principle, related to parallelism, says that deterministic reductions can be executed in parallel. Thus, the “parallelism part” of the basic Andorra principle, once isolated from the computation rule part, brings a basic principle to parallelism: that of the general
convenience of parallel execution of deterministic threads.

A fundamental conclusion stemming from the separation provided above is that parallel execution, as we have defined it, can at most provide a linear speedup. Furthermore, in the idealized case in which there are no task creation and scheduling overheads, then speedup is guaranteed if there is any parallelism being exploited.

Once the separation of parallelism issues from those that are related to computation rules has been discussed, let us now concentrate on another such “separation”: isolating the fundamental principles governing parallelism in the different models from the level of granularity at which they are applied. This means viewing the parallelizing principle involved as associated to a generic concept of thread, to be particularized for each system, according to the fundamental unit of parallelism used in such system.

As an example, and following these ideas, the fundamental principle of determinism used in the basic Andorra model can be applied to the goal-level independent and-parallel model (e.g. to the &-Prolog system). The basic unit of parallelism considered when parallelizing programs in the classical &-Prolog tools is the subtree corresponding to the complete resolution of a given goal in the resolvent. If the basic Andorra principle is applied at this level of granularity its implications are that deterministic subtrees can and should be executed in parallel (even if they are “dependent” in the classical sense).

Moving the notions of determinism in the other direction, i.e. towards a finer level of granularity, one can think of applying the principle at the level of bindings, rather than clauses, which yields the concept of “binding determinism” of PNU-Prolog [128].

In fact, the converse can also be done, and is the basis of our argument regarding the wide applicability of the results of this thesis: the underlying principles of the goal-level independent model w.r.t. parallelism –basically the notion of independence– can be applied at the granularity level of the Andorra model and/or other models without any significant change and providing either richer models or insight into existing models. In order to do this it is quite convenient to look at the basic operations in the light of David Warren’s extended Andorra model [154].\(^1\) The extended Andorra model brings in the first place the idea of presenting the execution of logic programs as a series of simple, low level operations on and-or trees. In addition to defining a lower level of granularity, the extended Andorra model incorporates some principles which are related in part to parallelism and in part to computation rule related issues such as the incorporation of the above mentioned basic Andorra principle and the avoidance of re-computation of goals.

On the other hand the extended Andorra model also leaves several other issues rel-

\(^1\)This is understandable, given that adding independent and-parallelism to the basic Andorra model was one of the objectives in the development of its extended version.
atively more open. One example is that of when nondeterministic reductions may take place in parallel. One answer for this important and relatively open issue was given in the instantiation of the model in the AKL language [89]. In AKL the concept of “stability” is defined as follows: a configuration (partial resolvent) is said to be stable if it cannot be affected by other sibling configurations. In that case the operational semantics of AKL allow the non-determinate promotion to proceed. Note that the definition is, not surprisingly, equivalent to that of independence, although applied at a different granularity level to that used in our presentation. Unfortunately stability/independence is in general an undecidable property. However, applying the work previously developed in the context of independent and-parallelism and that developed in this thesis at this level of granularity provides sufficient conditions for stability/independence in many cases. The usefulness of this is underlined by the fact that the current version of AKL incorporates the relatively simple notion of strict independence (i.e. the absence of variable sharing) as its stability rule for the Herbrand domain. However, the presentation above clearly marks the way for incorporating more advanced concepts, such as non-strict independence and the concepts of independence that we have developed for constraint domains and in the presence of concurrent goals, as a sufficient condition for the independence/stability rule. Note that, as we have shown, when adding support for constraints to a system the traditional notions of independence are no longer valid and both new definitions of independence and sufficient conditions for it need to be developed. We believe that the view proposed herein allows the direct application of our general results concerning independence in constraint systems to several realms, such as the extended Andorra model and AKL.

Also, another way of moving the concept of independence to a finer level of granularity is to apply it at the binding level. This yields a rule which states that dependent bindings of variables should wait for their leftmost occurrences to complete (in the same way as subtrees wait for dependent subtrees to their left to complete in the standard independent and-parallelism model). Traditional strict independence, when applied at this level of granularity, is essentially equivalent to the underlying rule of the DDAS model. Thus, one could easily imagine applying the more advanced independence notions for LP and CLP presented in the thesis at the level of bindings, which would yield new versions of DDAS which would not require dependent bindings to wait for bindings to their left which are guaranteed to never occur, or for bindings which are guaranteed to be compatible with them, and which would correctly ensure no-slowdown for CLP programs.
8.2 Topics for Future Work

To conclude, we point out a few avenues for future work. One of the clear topics for the future is to evaluate the practical usefulness of the techniques developed in terms of actual performance improvement in an actual parallel constraint logic programming system, possibly with dynamic scheduling. This evaluation has not been performed in the thesis since there is no practical system of these characteristics available at this time. However, we expect such systems to become widely available in the short term. In particular, a constraint-based version of the &-Prolog system (called CIAO – Concurrent constraint Independence-based And/Or system [76]), based on a glass box approach to both constraints and concurrency [70], is now starting to be operational and will be a test vehicle for our ideas. The notions presented in this thesis, from the concepts of independence to the global analysis technology, are being instrumental in the development of such a system, in which a complete parallelizing compiler for CLP languages with dynamic scheduling is being developed.

Another issue that must be further studied is the characterization of existing and proposed solvers from the point of view of solver independence. A complementary and interesting approach taken by Bueno et al [18] making use of the fine-grained semantics of Montanari and Rossi [118], would be to have access to the low level transitions performed by the constraint solver. Then, the amount of work performed by the constraint solver in adding a particular constraint to a particular store, would become explicit, and it could be characterized in terms of search space, analogously as for the high level transitions. Also, extension of the concept of solver independence in languages with dynamic scheduling needs further study. It should be necessary to characterize the conditions under with the amount work involved in determining if a goal must become delayed or must be woken is not significantly affected by changing the store.

As another suggestion for future work, consider the fact that the proposed a-priori conditions for independence pay in efficiency their generality, since they were designed to be applicable to any constraint system. The suggestion is to study several other (more and more specialized) a priori conditions for particular CLP languages and domains, and also to evaluate the cost-performance tradeoff involved.

Another suggestion is to further improve the global analysis framework for dynamically scheduled languages, finding better conditions for avoiding non termination which have advantages either in performance or in reducing the loss of accuracy due to the widening operation. Also, it would be interesting to evaluate the applicability in practice of the conditions which allow parallelization when there exist delayed literals before the execution of the parallel goals.

Finally, a topic of much interest is to follow on the ideas of Section 8.1, study-
ing how the notions developed can be applied at different granularity levels. Note that, when applied at the ultimately fine granularity level, the notions of independence developed in the thesis involve ensuring the preservation of the consistency or inconsistency of constraint application to the store with respect to the sequential model. Parallelization using this notions and at this level of granularity can allow very high levels of parallelism. This notion is being explored further in [119, 18, 15] where very rich and powerful parallelization and sequentialization rules are derived from a “truly concurrent” semantics for LP, CLP, and CC languages.
Bibliography


[27] M. Codish, M. García de la Banda, M. Bruynooghe, and M. Hermenegildo. Goal
Dependent vs Goal Independent Analysis of Logic Programs. In F. Pfenning, ed-
itor, Fifth International Conference on Logic Programming and Automated Rea-
soning, number 822 in LNAI, pages 305–320, Kiev, Ukraine, July 1994. Springer-
Verlag.

Logic Programs. In Eighth International Conference on Logic Programming, pages

concurrent constraint logic programs. In Twentieth International Col.

[30] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and
M. Hermenegildo. Improving Abstract Interpretations by Combining Domains.
In Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based

[31] C. Codognet, P. Codognet, and M. Consini. Abstract Interpretation of Concurrent
Logic Languages. In North American Conference on Logic Programming, pages

Programs. In Fourth International Conference on Programming Languages, April

1991 International Conference on Logic Programming, pages 127–141. MIT Press,
June 1991.

[34] A. Colmerauer. Les gramaire de metamorphose. Technical report, Univ. D’aix-
Marseille, Groupe De Ia, 1975.


[36] J. S. Conery. The And/Or Process Model for Parallel Interpretation of Logic
Report 204.

[37] J. S. Conery. Binding Environments for Parallel Logic Programs in Nonshared


188


193


