[17] D. Turner. Functional Programming and Communicating Processes. *Lectures Notes in Computer Science, Volumen 259*, páginas 54-74. Springer Verlag, 1987.

[18] P. Wadler. Comprehending monads. *Proceedings ACM Conference on Lisp and Functional Programming*, Nice, Junio 1990.

[19] P. Wadler. The essence of functional programming. En *19th Symposium on Principles of Programming Languages*. ACM, 1992.

Methods for Automatic Compile–time Parallelization of Logic Programs: The MEL, UDG, and CDG Algorithms Revisited*

F. Bueno

bueno@fi.upm.es
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - Spain
Phone: (9)1-336-7448 Fax: (9)1-352-4819

Abstract

This paper presents a study of three different algorithms for the parallelization of logic programs based on compile–time detection of independence among goals and the use of fork and join constructs. The study starts by revisiting their definitions as proposed in the literature, proposing extensions to them in various ways, and suggesting different possible improvements on these extensions. Comparison of the different alternatives is carried over mainly through examples, discussing the trade-offs involved.

1 Introduction

Independence has been since some time identified as a desirable principle to allow parallel execution in (constraint) logic programs. The importance of independence lies in the fact that it has been shown to ensure both correctness and efficiency of the parallel execution [10]. By this we mean that the parallel execution returns (at least) the same solutions as the sequential execution, that all those solutions are correct, and that the "no-slowdown" property is preserved, i.e. that, under certain assumptions, parallelized programs will at the minimum not run any slower than their sequential counterparts (and, hopefully, run much faster). These properties are arguably quite desirable in practice.

The rule is generally applied at the level of granularity of goals (i.e. *Goal* Independence): the units which are determined to be independent and executed in parallel are the subtrees associated with a given set of goals in the resolvent. Furthermore, it is usually also applied in a *restricted* (i.e. fork and join) fashion: computation is forked in the resolvent into the separated subtrees which will execute in parallel, and resumed after completion by joining the answers returned. The rule can however

*This work was funded by ESPRIT project 6707 "*ParForce*" and CICYT project TIC93-0976-C.

be applied again to the new goals in subsequent resolvents thus possibly generating additional parallelism within each subtree. This is a well understood concept already present in different ways in the work of [5, 9, 12, 13] and others, which is now known as Independent And-Parallelism (IAP). *Strict* Independence [10] is arguably the most often used property in selecting goals to be run in parallel in the context of the independent and-parallelism model. *Non-Strict* Independence [10] extends the previous one and relaxes it. Search Independence [6] and its related concepts generalize the notions in the former two in several ways.

It turns out that independence is in general undecidable at compile–time since it depends on run–time instantiations of variables. On the other hand, checking independence of a set of goals in the resolvent at run–time is not always straightforward, and in general implies sufficient conditions which can be checked prior to the execution of the goals involved. These conditions, in turn, should be accomodated by (hopefully inexpensive) checks. But this is not the only source of run–time overhead: there is also a cost involved in having to consider all possible combinations of goals in all resolvents for parallelization. An interesting way of reducing both sources of overheads is to mark at compile–time selected literals in the program for parallel execution and, when independence cannot be determined statically, generate parallelization tests which will minimize the independence checking overhead for the goals arising from such literals. These are the motivating ideas behind the standard IAP model [9, 12, 10]. In general, given a collection of literals in the program which have been selected for parallel execution, we would like to be able to generate at compile–time a condition *i_cond* which, when evaluated at run–time, would guarantee the 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.

In the following, we will summarize the above process for the case of strict independence. Consider the set of conditions which includes "true", "false", or any set, interpreted as a conjunction, of one or more of the tests $ground(x)$, $indep(x,y)$, where x and y can be goals, variables, or terms in general. Let $ground(x)$ be true when x is ground and false otherwise. Let $indep(x,y)$ be true when x and y do not share variables and false otherwise. If the literals $g_1, \ldots, g_n$ to be parallelized are considered in isolation, rather than as part of a program, an example of a correct *i_cond* is $\{ground(x) | \forall x \in SVG\} \cup \{indep(x,y) | \forall (x,y) \in SVI\}$, where SVG and SVI are defined as follows:

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

It is easy to see that in general a groundness check is less expensive than an independence check, and thus a condition, such as the one given, where some independence checks are replaced by groundness checks is obviously preferable. It is important to point out 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: $SVI = \{(V,W)$ such that $\exists i,j, i < j$, with $V = vars(g_i) - SVG$ and $W = vars(g_j) - SVG\}$. In many implementations this "compacted" set of pairs is less expensive to check than that generated by the previous definition of SVI.

If the above condition is satisfied the literals are strictly independent for any possible substitution, thus ensuring that the goals resulting from the instantiations of such literals will also be strictly independent. However, when considering the literals involved as part of a clause and within a program, the test 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 program analysis in the process of automatic parallelization.

Once there are sufficient conditions to check the independence of goals, there remains the problem of deciding which combinations of goals are to be run in parallel, and *annotating* them with suitable checks. In [14] three algorithms were defined to perform this task at compile–time for Strict (and restricted) IAP, namely **MEL**, **CDG**, and **UDG**. These algorithms have been since modified and extended in various ways. We present these extensions in the following. The algorithms are studied w.r.t. the task they were designed for, automatic parallelization based on independence, disregarding the particular notion of independence used for building the run–time checks. However, examples are given for the case of Strict IAP.

2 Automatic Parallelization

We consider the &-Prolog language as a vehicle for expressing goal–level, restricted, independent and-parallelism.[1] For our purposes, &-Prolog is essentially Prolog, with the addition of the parallel conjunction operator "&" (used in place of "," –comma– when goals are to be executed in parallel), and a set of parallelism-related builtins, which includes the groundness and independence tests already described. For syntactic convenience an additional construct is also provided: the *Conditional Graph Expression (CGE)*. A CGE has the general form (*i_cond* => $goal_1$ & $goal_2$ & ... & $goal_N$) where *i_cond* is a sufficient condition for running $goal_i$ in parallel under the appropriate notion of independence, in our case the one described in the previous section. &-Prolog if-then-else expressions and CGEs can be nested in order to create richer execution graphs. One advantage of &-Prolog is that it respects Prolog syntax and thus it is possible to view the parallelization process as a source to source transformation. Such a transformation is called an *annotation*.

Annotators are thus concerned with identifying the opportunities for parallel execution in programs. The aim of the annotation process is to partition the original program into processes which are independent and which can thus be safely run in parallel. This task is performed as source to source transformations of the program

[1] Note, however, that the &-Prolog language is rich enough to express *unrestricted* and-parallelism, and at levels of granularity other than the goal level.

in which each clause is annotated with parallel expressions and checks which encode the notion of independence used.

Example 1 *A possible CGE for the goals* `a(w), b(x,y), c(y,z)` *under strict independence would be:*

```
..., a(w), (ground(y), indep(x,z) => b(x,y) & c(y,z)), ...
```

An alternative would be:

```
..., (indep(w,x), indep(w,z), indep(x,z), ground(y)
          => a(w) & b(x,y) & c(y,z)), ...
```

and yet another alternative:

```
..., (indep(w,[x,y]) -> (a(w) & b(x,y), c(y,z))
          ; a(w), (ground(y), indep(x,z) => b(x,y) & c(y,z))), ...
```

and so on.

Note that, given a clause, several different annotations are possible. Different heuristic algorithms implement different strategies to select among all possible parallel expressions for a given clause.

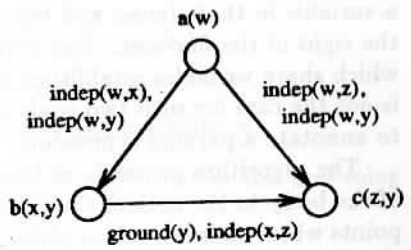
The annotation process is divided into two subtasks. The first one is concerned with identifying the dependencies between each two goals in a clause and the minimum number of tests for ensuring their independence, based on the sufficient conditions applicable. The second one is concerned with the core of the annotation process, namely to apply a particular strategy in order 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 rest of this section we introduce the first step and a simplification function for tests which can be applied in both steps, when information about the program is available, to reduce the checks.

It must be taken into account that, in general, side–effects cannot be allowed to execute freely in parallel with other goals. Although smarter approaches can be taken, no side–effect builtin or procedure calling a side–effect builtin is considered for parallelization by the (instances of the) algorithms we have studied. Also, some limited knowledge regarding the granularity of the goals, in particular the builtins, is used. As a result the only kind of builtins allowed to be run in parallel are certain meta–calls.

2.1 Identifying Dependencies

The dependencies between goals can be represented in the form of a dependency graph. 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 sufficient 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."

Example 2 *Consider a clause body with the goals of Example 1. The left-to-right precedence relation for the goals in the body of the clause can be represented using a directed, acyclic graph in which, using the rules described in Section 1, we associate with each edge which connects a pair of literals the sufficient condition for their strict independence, thus resulting in the dependency graph attached.*



2.2 Simplifying Dependencies

The annotation process can be improved by using compile–time information provided either by the user or by an analyzer. Local and global analyses have been defined for this purpose [1]. The improvement is then based on identifying tests which are ensured to either fail or succeed based on this information: if a test is ensured to succeed, it can be reduced to true; if a test is ensured to fail, it can be reduced to false. If the condition labeling an edge is reduced to true, the edge can be eliminated. On the contrary, if it is reduced to false, the edge becomes unconditional.

For any clause C, the information actually known at every program point i in C can be expressed in what we call a *domain of interpretation* GI: a subset of the first order logic theory, such that each $\kappa \in GI$ defined over the variables in C is a set of formulae (interpreted as a conjunction) representing facts about the variables of C.

For any program point i of a clause C where a test T_i on C variables is checked, the simplification process of such test, based on an element $\kappa_i \in GI$ defined over the variables in C, is defined by refining T_i to yield $T_i' = improve(T_i, \kappa_i)$, where:

$$improve(T_i, \kappa_i) = \begin{cases} if & \exists t \in T_i \; s.t. \; \kappa_i \vdash \neg t & then \; false \\ elseif & \kappa_i \vdash T_i & then \; true \\ else & for \; some \; t \in T_i & \{t\} \cup improve(T_i \setminus \{t\}, \kappa_i \cup \{t\}) \end{cases}$$

It is important to note that there is an implicit restriction on the selection of $t \in T_i$ in the above definition of *improve*: the order in which t is selected can influence the result of the *improve* function. Consider $\kappa_i = \{ground(x) \rightarrow ground(y)\}$ and $T_i = \{ground(x), ground(y)\}$, then, by selecting first $ground(y)$ the final result is $T_i' = T_i = \{ground(x), ground(y)\}$, whereas by selecting first $ground(x)$ the final result is $T_i' = \{ground(x)\} \subset T_i$, which is simpler. Thus, at least a restriction on the selection to first pick up conditions which do not appear as consequents in any atomic formula of κ_i should be considered. Also, for strict independence, a restriction to pick up groundness checks before independence checks can be considered (this one justified by the lower cost of groundness checks w.r.t. independence checks).

3 The MEL Algorithm

This algorithm is based on a heuristic which tries to find out points in the body of a clause where it can be split into different expressions [14]. One such point occurs where

a new variable appears. Consider a goal in a clause which has the first occurrence of a variable in that clause, and this variable is used as an argument of another goal to the right of the first one. The condition in Strict IAP which must hold for two goals which share variables establishes that these variables must be ground; obviously this is not the case for such two goals, and thus this is a point where it is not appropriate to annotate a parallel expression.

The algorithm proceeds in this manner from right to left, i.e. from the last goal of the body to the neck of the clause. The clause body is then broken into two at the points where shared new variables appear, and a parallel expression (a CGE) built for the right part of the sequence split. In proceeding backwards the underlying intention is to allow capturing the longest parallel expressions possible. An equivalent heuristic will proceed forwards but split the body at the second occurrences of new variables.

Example 3 *Consider the clause* `h(X) :- p(X,Y), q(X,Z), r(X), s(Y,Z).` *The original definition of* **MEL** *will compile it into the following &-Prolog parallel expression:*

```
h(X) :- ground(X) ⇒ p(X,Y) & q(X,Z),
        indep(X,Y), indep(X,Z) ⇒ r(X) & s(Y,Z).
```

Note that the body is split at `q(X,Z)` *and not at* `p(X,Y)`*, the largest expression being achieved in this way. Note also that the first CGE does not have the condition* `indep(Y,Z)` *since this condition is automatically satisfied by virtue of the fact that* `Z` *is a new variable.*

3.1 Generalising MEL

Recall the considerations in Section 2 on the process of automatic parallelization. Let a CDG be built for each clause in the program being annotated. We now propose a new version of the **MEL** algorithm, based on CDGs, in which its main heuristic is made independent of strict independence. Thus, the algorithm can be now applied to different notions of independence.

Let $C = H :- B$ and $B = \langle B_1, \ldots, B_n \rangle$. Define $\mathcal{K} = \{\mathcal{K}(B_i)\}$ where $\mathcal{K}(B_i) \in GI$ is the set of the relevant items of information for the independence notion under consideration which are known to be true *before* executing B_i. Define $\mathcal{I} = \{\mathcal{I}(B_i, B_j) \mid i \leq j\}$, where $\mathcal{I}(B_i, B_j)$ are the sets of conditions such that B_i and B_j are independent, which are already simplified w.r.t. $\mathcal{K}(B_i)$, i.e. these are the labels of the CDG for the clause C. The algorithm starts with a sequence B of goals (initially the body of the clause C) and computes its correspondent parallel expression D as follows:

1. Let $B = \langle B_1, \ldots, B_q \rangle$. Find the largest p such that for some $B_i, p < i \leq q$: $\mathcal{I}(B_p, B_i) = \{false\}$.
   If there is no such p, then let $p = 0$, $B1 = \langle\rangle$ and $B2 = B$. Else, let $B1 = \langle B_1, \ldots, B_p \rangle$ and $B2 = \langle B_{p+1}, \ldots, B_q \rangle$.

2. Let $IConds = improve(\bigcup_{\substack{p<i\leq q \\ i<j\leq q}} \mathcal{I}(B_i, B_j), \mathcal{K}(B_{p+1}))$ be the simplified set of conditions for goals in $B2$ to be independent.

3. Therefore, the &-Prolog parallel expression corresponding to $B2$ is

$$D2 = ((\bigwedge_{x \in ICond_s} goal(x)) \Rightarrow B_{p+1} \& \ldots \& B_q)$$

   where $goal(x)$ is the &-Prolog goal which satisfies the condition x.

4. If $B1 = \langle\rangle$, then $D = D2$. Else, $D = \langle D1, D2 \rangle$ where $D1$ is the &-Prolog parallel expression corresponding to $B1$.

Example 4 *Consider the clause in Example 3. Under strict independence, and with a simple local analysis, we have the following:*

$\mathcal{K}$(`p(X,Y)`) = {`free_not_aliased(Z)`}
$\mathcal{I}$(`p(X,Y)`, `q(X,Z)`) = {`ground(X)`, `indep(Y,Z)`}
$\mathcal{K}$(`q(X,Z)`) = {`free_not_aliased(Z)`}
$\mathcal{I}$(`q(X,Z)`, `r(X)`) = {`ground(X)`}
$\mathcal{I}'$(`q(X,Z)`, `s(Y,Z)`) = {`ground(Z)`, `indep(X,Y)`}
$\mathcal{I}$(`q(X,Z)`, `s(Y,Z)`) = {`false`}
$\mathcal{K}$(`r(X)`) = $\emptyset$
$\mathcal{I}$(`r(X)`, `s(Y,Z)`) = {`indep(X,Y)`, `indep(X,Z)`}

So C will be compiled into the following &-Prolog parallel expression:

```
h(X) :- ground(X) ⇒ p(X,Y) & q(X,Z),
        indep(X,Y), indep(X,Z) ⇒ r(X) & s(Y,Z).
```

Note that the first CGE does not have the condition `indep(Y,Z)` *since this condition is automatically satisfied by virtue of the fact that* `free_not_aliased(Z)` $\in \mathcal{K}$(`p(X,Y)`).

The expressions that the new definition of **MEL** produces for the case of strict independence are always the same as those which the original algorithm would have produced. The benefits of the new definition consist in making **MEL** applicable to non-strict (and possibly search) independence.

4 The UDG Algorithm

This algorithm starts with a graph $G(V, E)$ which is a UDG, where all dependencies are unconditional. The algorithm seeks to maximize the amount of parallelism possible under these dependencies. This is achieved if for any two goals for which a dependency is not present, they are annotated to be run in parallel — thus, no loss of parallelization opportunities occurs. For this, the transitive dependency relations among goals, represented by the graph edges, are considered, and conditions upon these established. Recall in the following that the UDG is transitively closed, thus it holds that $\forall x, y \in V \cdot x\ dep^*\ y \Leftrightarrow (y, x) \in E$.

The **UDG** algorithm works as follows [14]. It starts with the set of independent goals $I = \{p \in V \mid \forall x \in V \neg\exists(x, p) \in E\}$, those which do not have incoming edges. A

set of partitions $PP = \{P \in 2^I \mid \forall p \in P \cdot \exists x \in V \cdot x\ dep^*\ p\}$ is then built, so that there is at least one goal in $V - I$ for each of these partitions P which depends on all elements of P. These goals are grouped together so that $\forall P_i \in PP \cdot Q_i = \{x \in V \mid \forall p \in P_i \cdot x\ dep^*\ p\}$. In this context, no loss of parallelism can occur when converting the graph into a linear (parallel) expression, if and only if $\forall P_1 P_2 \in PP$,

- $P_1 \cap P_2 = \emptyset \vee P_1 \cap P_2 = P_1 \vee P_1 \cap P_2 = P_2$, and
- $P_1 \cap P_2 = P_1 \Rightarrow \forall q_1 \in Q_1, q_2 \in Q_2 \cdot q_2\ dep^*\ q_1$

Under such conditions, the corresponding parallel expression $exp(V)$ is built up by combining pairwise the subexpressions from the following rules, $\forall P_1 P_2 \in PP$ if:

- $P_1 \cap P_2 = \emptyset$ then a subexpression is $exp(P_1 \cup Q_1) \& exp(P_2 \cup Q_2)$
- $P_1 \cap P_2 = P_1$ then a subexpression is $exp(P_1 \cup Q_1) \& exp(P_2 - P_1), exp(Q_2)$

Note that no loss of parallelism occurs using the expressions implied by these two rules, because for a transitively closed graph it holds that $\forall P_1 P_2 \in PP$:

- $P_1 \cap P_2 = \emptyset \Rightarrow \forall q_1 \in Q_1, q_2 \in Q_2 \cdot \neg q_1\ dep^* q_2 \wedge \neg q_2\ dep^* q_1$
- $P_1 \cap P_2 = P_1 \Rightarrow \forall q_1 \in Q_1, q_2 \in Q_2 \cdot \neg q_1\ dep^* q_2$

Example 5 *Consider the clause* `h :- p(X), q(Y), r(X), s(X,Y).` *There are unconditional dependencies for* `r(X)` *on* `p(X)` *and* `s(X,Y)` *on* `p(X)` *and* `q(Y)`*, and a dependency (labelled* `ground(X)` *for* `s(X,Y)` *on* `r(X)`*.* **UDG** *will regard the latter as unconditional and compile this clause (applying the second rule above) to:*

```
h :- ( p(X), r(X) ) & q(Y), s(X,Y).
```

4.1 Extending UDG

In real program clauses, usually bodies are transformed to UDGs which are not transitively closed. We will consider extending the algorithm to these cases. From the definition of the partitions in PP, it always holds that $\forall P_1 P_2 \in PP \cdot P_1 \neq P_2$ and:

(1) $P_1 \cap P_2 = \emptyset$, or (2) $P_1 \cap P_2 = P_1$, or $P_1 \cap P_2 = P_2$, or

(3) $P_1 \cap P_2 = P$ s.t. $P \neq \emptyset, P \neq P_1, P \neq P_2$

The **UDG** algorithm finds out the best linearization of the dependency graph in such a way that no loss of parallelism occurs. For this to be possible, the body of a given clause must fit into either case (1) or a special sub–case of case (2). In order to extend the algorithm to deal with all possible cases, the following possible graph linearizations have to be considered: $\forall P_1 P_2 \in PP$, if

1. $P_1 \cap P_2 = \emptyset$ then $exp(P_1 \cup Q_1) \& exp(P_2 \cup Q_2)$
2. $P_1 \cap P_2 = P_1$ and

   (a) $\forall q_1 \in Q_1, q_2 \in Q_2 \cdot q_2\ dep^*\ q_1$ then $exp(P_1 \cup Q_1) \& exp(P_2 - P_1), exp(Q_2)$

   (b) $\forall q_1 \in Q_1, q_2 \in Q_2 \cdot \neg q_2\ dep^*\ q_1$ then

      i. at the loss of parallelism between Q_1 and Q_2

      $$exp(P_1 \cup Q_1) \& exp(P_2 - P_1), exp(Q_2)$$

      ii. at the loss of parallelism between Q_1 and $P_2 - P_1$

      $$exp(P_1) \& exp(P_2 - P_1), exp(Q_1) \& exp(Q_2)$$

   (c) $\exists q_1 \in Q_1, q_2 \in Q_2 \cdot q_2\ dep^*\ q_1$ then

      i. at the loss of parallelism between $q_2 \in Q_2$ and $q_1 \in Q_1$ s.t. $\neg q_2\ dep^*\ q_1$

      $$exp(P_1 \cup Q_1) \& exp(P_2 - P_1), exp(Q_2)$$

      ii. at the loss of parallelism between Q_1 and $P_2 - P_1$

      $$exp(P_1) \& exp(P_2 - P_1), exp(Q_1 \cup Q_2)$$

      iii. being $Q_{12} = \{q_1 \in Q_1 \mid \exists q_2 \in Q_2 \cdot q_2\ dep^*\ q_1\}$ and $Q_{11} = Q_1 - Q_{12}$

      $$exp(P_1 \cup Q_{12}) \& exp(P_2 - P_1), exp(Q_{11}) \& exp(Q_2)$$

      at the loss of parallelism between Q_{11} and $P_2 - P_1$ and also between $q_2 \in Q_2$ s.t. $\forall q_1 \in Q_1 \cdot \neg q_2\ dep^*\ q_1$ and Q_{12}

3. $P_1 \cap P_2 = P \mid P \neq \emptyset, P \neq P_1, P \neq P_2$ then $exp(P_1 \cup P_2), exp(Q_1 \cup Q_2)$ at the loss of parallelism between $q_2 \in Q_2$ and $p_1 \in P_1 - P$ and also $q_1 \in Q_1$ and $p_2 \in P_2 - P$

The original **UDG** algorithm deals with cases 1 and 2a. The natural extension of the algorithm to be able to deal with the whole of case 2 is to make it force the assumption that the required condition in case 2a holds and let it behave as in this case. This leads the extended algorithm to take into account cases 2(b)i and 2(c)i. To make the extension complete, it has to also deal with case 3, for which the partitions involved are considered as a single one: a new partition is built up from the union of the other ones. Note that each of these extensions implies a loss of parallelism.

4.2 Improving UDG

The proposed extension to the **UDG** algorithm seems to be the most natural, although at the cost of having to drop the objective of maximizing possibilities for parallelism. Furthermore, when it is considered w.r.t. the execution efficiency of the parallel expressions it obtains, it turns out that it may be more profitable to perform the extension in a different way.

This can be seen with a simple experiment. Consider all the possible parallel expressions listed above for a situation like case 2 and let $P_1 = \{p_1\}$, $P_2 = \{p_1, p_2\}$,

$Q_1 = \{q_{11}, q_{12}\}$ and $Q_2 = \{q_2\}$ be the minimum sets which fulfill the conditions in that situation. Suitable goals can be defined so that the conditions in Case 2 of previous subsection hold, yielding the following expressions:

2(b)i $(p_1, q_{11}, q_{12})\&p_2, q_2$ 2(c)i $(p_1, q_{11}, q_{12})\&p_2, q_2$ 2(c)ii $p_1\&p_2, q_{11}\&q_{12}, q_2$

2(b)ii $p_1\&p_2, (q_{11}, q_{12})\&q_2$ 2(c)iii $(p_1, q_{12})\&p_2, q_{11}\&q_2$

Let us now consider giving to each of the goals a measure of an upper bound on its granularity and computing all possible combinations of granularities of all the goals. The efficiency of each of the possible parallel expressions proposed above can be computed and are shown in Table 1. There, the percentage of combinations where each parallel expression behaves best is shown. The percentage includes the situations where all the expressions behave the same, thus the total percentage can add up to more than 100%.

Case	Maximum Goal Grain									
	1	2	3	4	5	6	7	8	9	10
2(b)i	0	18	22	23	24	24	24	24	24	24
2(b)ii	100	100	97	95	94	93	92	92	91	91
2(c)i	0	12	15	16	16	17	17	17	17	17
2(c)ii	100	100	96	93	92	91	90	89	88	88
2(c)iii	0	12	12	11	11	10	10	10	10	9

Table 1: Performance test for parallel expressions

From the table, it is clear that the best parallelization strategy corresponds to the second option in both cases. This is due to the fact that this strategy performs a better load balancing of parallel tasks with goals which are already balanced (i.e. have almost the same granularity, as with maximum grain of 1 or 2) or for which the differences in grain are not high. When a bigger difference is allowed (increasing the maximum permitted goal grain) the average efficiency of this strategy lowers a bit, while the first strategy (that pointed out in the previous section) progressively behaves better, but in any case the asymptotic values seem to stabilize.

Therefore, the best parallelization strategy, which should be used to extend the **UDG** algorithm, is that of cases 2(b)ii and 2(c)ii. In both cases the strategy is the same: a parallel expression is built up for the P_i goal partitions and separately for the Q_i ones and these two are annotated for sequential execution.

It is worth noting that this result points out the importance of having granularity information on the goals being annotated, so that the annotators could take granularity into consideration in the load balancing algorithms. Unfortunately, having good measures for the granularity of goals is a difficult task.[2] In the absence of information on granularity, the parallelization strategy herein presented should be pursued.

On the other hand, it turns out that an algorithm exploiting this strategy has to locally consider the goals to be annotated. When the whole of a clause is considered

[2] Although interesting progress is being made recently — see [7, 8].

(as it is the case in an UDG), many different instances of the possible cases listed above arise. If a grouping of goals such as the partitions the algorithm makes is done, the strategy does not perform as intended. Note that the cases in which this strategy is based cause sequentialization of all elements of $Q_1 \cup Q_2$ w.r.t. all elements of $P_1 \cup P_2$. Thus, it suggests parallel expressions similar to those which will be obtained with the **UDG** original strategy if all elements of $Q_1 \cup Q_2$ depend on all elements of $P_1 \cup P_2$. Therefore, the new strategy suggests a grouping of partitions which when applied to a whole clause derives in too many "imaginary" dependencies being created. Therefore, this strategy should be applied in a goal–to–goal fashion, incrementally creating the required dependencies based on a pairwise consideration of goals. A similar approach to an algorithm in this style is presented in [4].

5 The CDG Algorithm

This algorithm is quite close to the previous one. In this case conditional dependencies present in a CDG $G(V, E)$ are considered. To do this, all possible states of computation which can occur w.r.t. the conditions present in the graph are considered, and the body goals annotated in the best parallel expressions achievable under these conditions. The algorithm, as originally designed [14], starts with the same set I of independent goals as above. The main difference relies in that goals depending unconditionally on goals in I are not coupled to them (i.e. the close relation upon each P_i and corresponding Q_i in the **UDG** algorithm is not followed here). On the contrary, the **CDG** algorithm focuses on the conditional dependencies in the graph.

Consider the set $D = V - I$ of dependent goals. The sets of conditions other than "false" in labels of edges between goals in I and goals in D, $IConds = \{label((p, x)) \neq false \mid (p, x) \in E, p \in I, x \in D\}$, and in labels of edges among goals in D, $DConds = \{label((x, y)) \neq false \mid (x, y) \in E, x, y \in D\}$ are built. The algorithm proceeds by incrementally building up the parallel expression exp as follows, let $I = \{p_1, \ldots, p_n\}$:

(1) if $D = \emptyset$ then exp is $\langle p_1\& \ldots \&p_n \rangle$

(2) if $D \neq \emptyset$, $IConds = DConds = \emptyset$ then exp is built using **UDG**

(3) if $D \neq \emptyset$, $IConds = \emptyset$, $DConds \neq \emptyset$ then exp is $\langle p_1\& \ldots \&p_n, exp_1 \rangle$, where exp_1 is recursively computed for $G(D, E_1)$ being $E_1 = E \setminus \{(p, x) \in E \mid p \in I\}$

(4) if $D \neq \emptyset$, $IConds \neq \emptyset$ then exp is built up from the boolean combinations of the elements of $IConds$

For each boolean combination C the graph $G(V, E)$ is updated as if the conditions in C hold, that is, all conditions in labels of edges of E which are implied by elements of C are deleted and all labels with conditions which are incompatible with some element of C rewritten into "false." Note that an edge can be removed if its label becomes void (i.e. "true"). In [14] an algorithm to perform this updating in the particular case of Strict IAP is presented. The parallel expressions coming out of recursively applying the **CDG** algorithm after this updating are annotated as if–then–elses and combined in a simplified form.

Example 6 *Consider a clause* `p(W,X,Y,Z) :- W is X+1, a(W), b(X,Y), c(Y,Z)` *with the goals of Example 1.* **CDG** *will consider all possible alternatives and yield:*

```
p(W,X,Y,Z) :- W is X+1, ( ground(Y) -> a(W) & b(X,Y) & c(Y,Z)
                        ; a(W) & (b(X,Y), c(Y,Z)) ).
```

whereas **UDG** *will yield:*

```
p(W,X,Y,Z) :- W is X+1, a(W) & (b(X,Y), c(Y,Z)).
```

and **MEL** *will annotate it as:*

```
p(W,X,Y,Z) :- W is X+1, ground(Y) => a(W) & b(X,Y) & c(Y,Z).
```

From the examples it can be thought that the heuristic applied by **CDG** is the best of the three presented. However in real cases and without accurate information, the number of tests created by **CDG** may yield significant overhead, making the other two approaches more practical (see [2]).

5.1 Improving CDG

The algorithm is focussed on annotating all possible parallel expressions which can be exploited based on the different situations which may occur at execution–time. It avoids dealing with unconditional dependencies and focusses on conditions present in the graph. Thus, in case (3) of the algorithm, an unconditional parallel expression is built for elements in I followed sequentially by another expression recursively computed for the rest of the goals. No consideration is done in this case regarding the unconditional dependencies which could occur for other goals on goals in I. The **UDG** algorithm, on the other hand, does this, and groups goals depending unconditionally on those of I together and with those on which they depend, building an expression for the different groups of goals made. An extension of the **CDG** algorithm in this direction will extract from the CDG the subgraph of unconditional dependencies on goals of I (for the case mentioned above) and behave as **UDG**, then returning back with an updated graph to the original algorithm. This extension will allow a one–to–one correspondence between both algorithms, so that expressions built up by **UDG** will always be the worst case subexpression of that built by **CDG** (as in Example 6).

Example 7 *On the contrary, consider the clause of Example 5. The original* **CDG** *algorithm will compile it to a very different expression from the one in that example:*

```
h :- p(X) & q(Y), ground(X) => r(X) & s(X,Y).
```

Further extensions of the **CDG** algorithm in the same direction can be done. The heuristic that this algorithm exploits of considering all combinations of conditions can be replaced by the **UDG** strategy of partitioning the graph into strongly dependent groups of goals. In this case the effect would be that the parallel expressions being annotated will look like conditional expressions nested inside unconditional ones. This turns out to be a different algorithm than **CDG**: it will consist in the same

UDG algorithm performing not only on unconditional edges of the graph but also on conditional ones, but in turn annotating these latter with the required conditions.

An algorithm in this style will work as follows. It will compute partitions as in **UDG**, but, unlinke this one, the partitions will always be disjoint. Thus, an unconditional parallel expression can be built for the subexpressions arising from recursively applying the algorithm to these partitions. Let $I = \{p \in V \mid \forall x \in V \neg\exists(x,p) \in E\}$ be the set of independent goals, as in **UDG**. We build the set of partitions $SS = \{S(p) \mid p \in I\}$ by transitively following the dependency links (but without transitively closing the graph $G(V,E)$):

$$\forall p \in I \cdot S(p) = \{p\} \cup \{x \in V \mid x\ dep^* p \wedge \nexists p' \in I \cdot x\ dep^* p'\}$$

Starting with graph $G(V,E)$, and given $S \subseteq V$, let $restrict(S,E) = \{(x,y) \in E \mid x,y \in S\}$. Let $update(C,E)$ denote a function which updates a set E of labelled edges w.r.t. a condition C, as in the **CDG** algorithm. Let also $select(L)$ denote a function which selects a condition from a set L of them. The new algorithm can be summarized [2] in the two following steps which build the expression $exp(V,E)$:

- if $\exists!\ S \in SS$ then $exp(V,E)$ is $\langle cond \Rightarrow exp(S, update(cond, restrict(S,E)))\rangle$ where $cond = select(\{label((x,y)) \mid x,y \in S\})$
- else, $SS = \{S_1,\ldots,S_n\}$, then $exp(V,E)$ is $\langle exp(S_1,E_1)\&\ldots\&exp(S_n,E_n)\rangle$ where $\forall i \in [1,n] \cdot E_i = restrict(S_i,E)$

Example 8 *Consider the clause* `h(X) :- p(X), q(Y), r(X), s(Y).` *There is an unconditional dependency for* `s(Y)` *on* `q(Y)` *and a dependency labelled with* `ground(X)` *for* `r(X)` *on* `p(X)`. *While* **CDG** *will annotate it as:*

```
h(X) :- ground(X) -> p(X) & r(X) & ( q(Y), s(Y) )
                  ; ( p(X), r(X) ) & ( q(Y), s(Y) ).
```

and **UDG** *will annotate it as:*

```
h(X) :- ( p(X), r(X) ) & ( q(Y), s(Y) ).
```

which is the worst case subexpression of the expression above; the new **CDG** *algorithm will annotate it as:*

```
h(X) :- ( ground(X) => p(X) & r(X) ) & ( q(Y), s(Y) ).
```

In this rather simple example, the last expression is equivalent to the expression of **CDG**. However, this is not always the case, and the new algorithm behaves better for clauses with big bodies (which sometimes pose serious problems to **CDG** — see [2]). The reason for this is that the new algorithm behaves in a stepwise manner, first allowing unconditional parallelism to be annotated, and then postponing the consideration of the conditions until no more unconditional parallelism can be exploited. Note that there is an implicit choice in the definition of the *select* function, where there is room enough for different heuristics.

Finally, it is important to note that, although **CDG** and **UDG** are both inherently applicable to different notions of independence, they may not be valid in all cases. The reason is that both of them freely alter the original order of the goals in a clause when they have been found independent. This is always done under the assumption that being the goals independent, they can be run in parallel, and thus their order in the program text is not relevant. Although this can be true for strict independence, since it is a transitive notion, it is not always applicable in non–strict or search independence, where the conditions may depend on the order of the goals [10, 6].

6 Conclusions

We have studied three different algorithms for automatic parallelization of logic programs, previously proposed in the literature. Their definitions have been revised and completed in some cases, and several possible extensions have been suggested. The study points out the importance of considering different alternatives for parallelization, but designing good heuristics (probably based on information about goal granularity) to select among them. The role of compile–time available information, and thus program analysis, has also been identified. Current work demonstrates the usefulness of analysis for this purpose [1].

Although very interesting, it is out of the scope of the paper, and subject of current work, the comparison of the original and new definitions of the algorithms from the performance point of view. A comparison between the *different* algorithms can be found in [2]. Regarding future work, one important issue is the automatic exploitation of non–strict independence for which automatic parallelization technology is becoming available [4]. The applicability of the algorithms studied is currently being considered, and it seems feasible in some cases. Another important potential improvement may be to detect parallelism at other levels of granularity than the goal level used in our study, as suggested in [11, 3]. Finally, we also will have to consider extending our study to the automatic parallelization of CLP programs, using as a starting point the generalized notions presented in [11, 6].

Acknowledgements

The author will like to thank Manuel Hermenegildo, who has made possible the work reported in this paper. It would not have been possible either without the collaboration of Kalyan Muthukumar, who defined the original algorithms. Thanks also to María José García de la Banda, for her brainstorming, Daniel Cabeza, for very useful discussions, and the CLIP group at U.P.M. for their comradeship.

References

[1] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. TR CLIP7/93.0, UPM, October 1993.

[2] F. Bueno, M. García de la Banda, and M. Hermenegildo. The MEL, UDG, and CDG Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-Parallelism: A Comparative Study. TR CLIP3/94.0, UPM, January 1994.

[3] F. Bueno, M. García de la Banda, M. Hermenegildo, U. Montanari, and F. Rossi. From Eventual to Atomic and Locally Atomic CC Programs: A Concurrent Semantics. TR CLIP1/94.0, UPM, January 1994. Presented at PPCP'94.

[4] D. Cabeza and M. Hermenegildo. Towards Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. TR CLIP5/92.1, UPM, August 1993.

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

[6] M.J.García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.

[7] S. Debray, P. López García, M. Hermenegildo, and N. Lin. Lower Bound Cost Estimation for Logic Programs. TR CLIP4/94.0, UPM, March 1994.

[8] S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.

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

[10] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 1994. To appear.

[11] M. Hermenegildo and the CLIP group. First steps towards a ciao-prolog system. In *Compulog Net Area Workshop on Parallelism and Implementation Technologies.* UPM, June 1993. Presented also at PPCP'94.

[12] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, August 1986.

[13] D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming*, pages 284–297, 1988.

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

Issues in Implementing Logic Languages

Peter Van Roy

Digital Equipment Corporation, Paris Research Laboratory
85 Avenue Victor Hugo, 92500 Rueil-Malmaison, France
vanroy@prl.dec.com

Abstract

How does one go about implementing a new logic language? What are the principles underlying simple and efficient implementations? What is the quickest way to build a fast system? How does one avoid reinventing the wheel? This talk provides answers to these and other questions. The talk is divided into three parts: high-level issues, Prolog implementation issues, and general implementation techniques. Examples are taken from various logic languages, including constraint languages. The talk provides a plethora of information that will be helpful to both the novice implementer and the experienced practitioner.

Optimizing Logic Programs with Finite-Domain Constraints *

Nai-Wei Lin

Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi, Taiwan 62107, R.O.C.

Abstract

Combinatorial problems are often specified declaratively in logic programs with constraints over finite domains. This paper presents several program transformation and optimization techniques for improving the performance of the class of programs with finite-domain constraints. The techniques include planning the evaluation order of body goals, reducing the domain of variables, and planning the instantiation order of variable values. The techniques are based on the information about the number of solutions of combinatorial problems. Since the decision versions of many combinatorial problems are NP-complete, if $P \neq NP$, there is no polynomial time algorithm for computing the number of solutions for combinatorial problems. This paper presents a simple greedy approximation algorithm for estimating the number of solutions for combinatorial problems over finite domains. Based on this simple algorithm, a class of flexible polynomial time algorithms is also developed.

1 Introduction

Many problems arising in artificial intelligence and operations research can be formulated as combinatorial problems. A combinatorial problem can be a constraint satisfaction problem or a combinatorial optimization problem. A constraint satisfaction problem involves a set of n variables, a domain of values, and a set of constraints on the variables. A solution to such a constraint satisfaction problem is an n-tuple of assignments of domain values to variables such that all the constraints in the problem are satisfied. Usually, either a single solution or all solutions are sought. A combinatorial optimization problem can be viewed as a constraint satisfaction problem, which is further subject to an objective function (a maximization or minimization function).

*This work was supported in part by the National Science Foundation of USA under grant number CCR-8901283 and the National Science Council of ROC under grant number NSC-83-0408-E-194-010.

Familiar problems that can be formulated as combinatorial problems include boolean satisfiability, subgraph isomorphism, graph coloring, etc.

Combinatorial problems are often specified declaratively in logic programs with constraints over finite domains. This paper will examine in particular the class of programs whose constraints involve only comparison operators ($=, \neq, >, \geq, <, \leq$). These logic programs are usually solved by tree search algorithms: backtracking [12], intelligent backtracking [1], forward checking, looking ahead, or branch and bound algorithms [4, 14]. The performance of the tree search algorithms depends heavily on the evaluation order among domain-value generators and among constraints [5], and the instantiation order of domain values to variables, that is, the order in which domain values are generated by generators [3]. For ordering the evaluation among generators and constraints, existing logic programming systems either dynamically order the evaluation of subgoals [1, 4, 14], which will inevitably incur some runtime overhead, or use very primitive information for static ordering [12], which only achieves restricted benefits. For ordering the instantiation of domain values to variables, no existing system supports this feature yet. This paper presents several program transformation techniques for statically planning the subgoal evaluation order and the instantiation order of domain values to variables. These techniques allow automatic transformation of declaratively specified programs into more efficient programs. The techniques are based on the information about the number of solutions of constraint satisfaction problems. Moreover, knowledge about the number of solutions of constraint satisfaction problems can also be applied to reduce the domain of variables.

The decision versions of many constraint satisfaction problems, such as boolean satisfiability, subgraph isomorphism, and graph k-colorability, that decide whether there exists a solution satisfying all the constraints in the problem are NP-complete [2]. Thus if P $\neq$ NP, there is no polynomial time algorithm for computing the number of solutions for constraint satisfaction problems. Rivin and Zabih [11] have proposed an algorithm for computing the number of solutions for constraint satisfaction problems by transforming a constraint satisfaction problem into an integer linear programming problem. If the constraint satisfaction problem has n variables and m domain values, and if the equivalent programming problem involves M equations, then the number of solutions can be determined in time $O(nm2^{M-n})$.

We are interested in finding algorithms for computing an upper bound on the number of solutions for constraint satisfaction problems over finite domains of discrete values. This paper presents a simple approximation algorithm based on the greedy method. Based on this simple algorithm, a class of flexible polynomial time approximation algorithms is also developed. A user can choose the appropriate algorithm according to specific efficiency and precision requirements. These algorithms are particularly useful for programs whose constraints are explicitly expressed as a set of constraints involving comparison operators. In general, it is very difficult to infer nontrivial number of solutions information for this class of programs without considering the net effects of the set of constraints. Estimation of the number of solutions for this class of programs has not been addressed in previous work [13, 15].

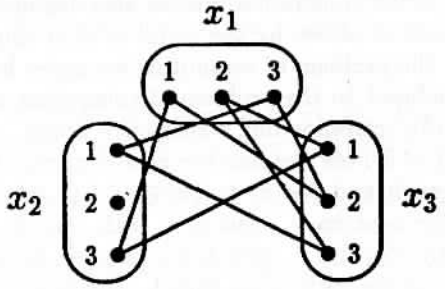


Figure 1: An example of a consistency graph

2 Number of Solutions of Constraint Satisfaction Problems

In this paper we only consider the following class of *binary constraint satisfaction problems* (binary CSPs): a binary CSP involves a finite set of variables $x_1, \ldots, x_n$, a finite domain of discrete values $d_1, \ldots, d_m$, and a set of unary or binary constraints on variables, namely, constraints involving only one or two variables. A solution to such a CSP is an n-tuple of assignments of domain values to variables such that all the constraints in the problem are satisfied. The approximation of n-ary CSPs (i.e., with constraints involving n variables) by binary CSPs is discussed in [10].

The set of constraints on variables can be represented as a graph $G = (V, E)$, called a *consistency graph*. Each vertex $v_{i,j}$ in V denotes the assignment of a value d_j to a variable x_i, written $x_i \leftarrow d_j$, for $1 \leq i \leq n$ and $1 \leq j \leq m$. There is an edge $\langle v_{p,g}, v_{q,h} \rangle$ between two vertices $v_{p,g}$ and $v_{q,h}$ if the two assignments $x_p \leftarrow d_g$ and $x_q \leftarrow d_h$ satisfy all the constraints involving variables x_p and x_q. The two assignments are then said to be *consistent*. The set of vertices $V_i = \{v_{i,1}, \ldots, v_{i,m}\}$ corresponding to a variable x_i is called the *assignment set* of x_i. The *order* of a consistency graph G is (n, m) if G corresponds to a CSP involving n variables and m domain values. Because two distinct values cannot be assigned to the same variable simultaneously, no pair of vertices in an assignment set are adjacent. Therefore, the consistency graph of a CSP involving n variables is an n-partite graph. As an example, the consistency graph for the problem (3-queens) with the set of variables $\{x_1, x_2, x_3\}$, the domain $\{1, 2, 3\}$ and the constraints $x_j \neq x_i$ and $x_j \neq x_i \pm (j-i)$, for $1 \leq i < j \leq 3$, is shown in Figure 1.

An *n-clique* of a graph G is a subgraph of G such that it has n vertices and its vertices are pairwise adjacent. Since a solution s to a CSP p involving n variables is an n-tuple of assignments of domain values to variables such that all the constraints in p are satisfied, every pair of assignments in s is consistent. Thus, if the constraints involve only conjunctions, then s corresponds to an n-clique of the consistency graph of p, and the number of solutions of p is equal to the number of n-cliques in the

consistency graph of p; if the constraints involve also disjunctions, then the number of solutions of p is bounded above by the number of n-cliques in the consistency graph of p. Therefore, the problem of computing an upper bound on the number of solutions of a CSP is reduced to the problem of computing an upper bound on the number of n-cliques in the corresponding consistency graph.

We will use $K(G,n)$ to denote the number of n-cliques in a consistency graph G. Let $G=(V,E)$ be a graph and $N_G(v)=\{w\in V \mid \langle v,w\rangle \in E\}$ be the *neighbors* of a vertex v in G. The *adjacency graph* of v with respect to G, $Adj_G(v)$, is the subgraph of G *induced* by $N_G(v)$, i.e., $Adj_G(v)=(N_G(v),E_G(v))$, where $E_G(v)$ is the set of edges in E that join the vertices in $N_G(v)$. The following theorem shows that the number of n-cliques in a consistency graph can be represented in terms of the number of $(n-1)$-cliques in the adjacency graphs corresponding to the vertices in an assignment set.

Theorem 2.1 *Let G be a consistency graph of order (n,m). Then for each assignment set $V=\{v_1,\ldots,v_m\}$, $K(G,n)=\sum_{i=1}^{m}K(Adj_G(v_i),n-1)$.*

Proof Let G_i be the subgraph of G induced by $N_G(v_i)\cup\{v_i\}$. Since no pair of vertices in V are adjacent, $K(G,n)=\sum_{i=1}^{m}K(G_i,n)$. Since v_i is adjacent to every vertex in $N_G(v_i)$, $K(G_i,n)=K(Adj_G(v_i),n-1)$. □

Using Theorem 2.1 directly, the computation will require exponential time $O(m^n)$. To make it practical, therefore, we need to find a way to combine the set of subgraphs $Adj_G(v_1),\ldots,Adj_G(v_m)$ in Theorem 2.1 into a graph H such that $K(H,n-1)$ is an upper bound on $K(G,n)$.

To derive an upper bound on $K(G,n)$ for a consistency graph G of order (n,m), we extend the representation of a consistency graph to a *weighted* consistency graph. A *weighted consistency graph* $G=(V,E,W)$ is a consistency graph with each edge $e\in E$ associated with a *weight*, $W(e)$, where $W: V\times V\rightarrow N$ is a function that assigns a positive integer to an edge $\langle u,v\rangle$ if $\langle u,v\rangle\in E$, and assigns 0 to $\langle u,v\rangle$ if $\langle u,v\rangle\notin E$. The aim is to use the weights to accumulate the number of n-cliques information.

The number of n-cliques, $K(G,n)$, in a *weighted* consistency graph G of order (n,m) is defined as follows. Let S be the set of n-cliques of G and $H=(V_H,E_H,W_H)\in S$ be an n-clique. We define $K(H,n)=min\{W_H(e)\mid e\in E_H\}$, and $K(G,n)=\sum_{H\in S}K(H,n)$. The intuition behind this formulation involves an operation on graphs and will become clear shortly. Let the weighted consistency graph corresponding to a consistency graph $G=(V,E)$ be $G'=(V,E,W)$, where $W(e)=1$, for all $e\in E$. Then we have $K(G,n)=K(G',n)$ by definition. Thus we can focus on weighted consistency graphs from now on.

We now define a binary operator $\oplus$, called *graph addition*, on two weighted consistency graphs. Let $G_1=(V,E_1,W_1)$ and $G_2=(V,E_2,W_2)$ be two weighted consistency graphs with the same set of vertices. Then $G_1\oplus G_2=(V,E_{1\oplus 2},W_{1\oplus 2})$, where $E_{1\oplus 2}=E_1\cup E_2$, and $W_{1\oplus 2}(e)=W_1(e)+W_2(e)$, for all $e\in E_{1\oplus 2}$. Graphs G_1 and G_2 are said to be the *component graphs* of $G_1\oplus G_2$. The intuition behind the definition of the number of n-cliques in a weighted consistency graph G is that for each n-clique

H of G, $K(H,n)=k$ implies that H appears in *at most* k component graphs of G. The following theorem shows the effect of graph addition on the number of n-cliques in the graphs.

Theorem 2.2 *Let $G_1=(V,E_1,W_1)$ and $G_2=(V,E_2,W_2)$ be two weighted consistency graphs of order (n,m). Then $K(G_1\oplus G_2,n)\geq K(G_1,n)+K(G_2,n)$.* □

The theorem is proved by a straightforward case analysis. We now define the weight of an edge in a weighted adjacency graph as follows. Let $Adj_G(v)=(V_A,E_A,W_A)$ be the weighted adjacency graph of v with respect to a weighted consistency graph $G=(V,E,W)$. For every edge $\langle u,w\rangle\in E_A$, we define $W_A(\langle u,w\rangle)=min(W(\langle v,u\rangle),W(\langle v,w\rangle),W(\langle u,w\rangle))$. This definition will lead to the same result for weighted consistency graph as Theorem 2.1 for consistency graph.

Theorem 2.3 *Let G be a weighted consistency graph of order (n,m). Then for each assignment set $V=\{v_1,\ldots,v_m\}$, $K(G,n)=\sum_{i=1}^{m}K(Adj_G(v_i),n-1)$.* □

By Theorems 2.2 and 2.3, we have the following theorem.

Theorem 2.4 *Let G be a weighted consistency graph of order (n,m). Then for each assignment set $V=\{v_1,\ldots,v_m\}$, $K(G,n)\leq K(\bigoplus_{i=1}^{m}Adj_G(v_i),n-1)$.* □

We are now ready to present a simple greedy algorithm for computing an upper bound on $K(G,n)$ for a weighted consistency graph G of order (n,m). The basic idea is to apply Theorem 2.4 repeatedly to a sequence of consecutively smaller graphs. By starting with the graph G, at each iteration, one assignment set is removed from the graph, and a smaller graph is constructed by performing graph addition on the set of adjacency graphs corresponding to the vertices in the removed assignment set. The algorithm is shown as follows:

```
    Algorithm n-cliques(G = (V, E, W), n, m)
1.  begin
2.      G_1 := G;
3.      for i := 1 to n − 2 do
4.          G_{i+1} := ⊕_{j=1}^{m} Adj_{G_i}(v_{i,j});
5.      od
6.      return Σ_{e∈E_{n−1}} W_{n−1}(e);
7.  end
```

Let $v=nm$ be the number of vertices in the graph and e the number of edges in the graph. The time complexity of the n-cliques algorithm can be shown to be $O(v(v+e))$.

We next investigate three methods for improving the estimation precision of the n-cliques algorithm: (I) *network consistency*, (II) *exact expansion*, and (III) *assignment memorization*.

(I). Since basic backtracking algorithms may incur significant inefficiency, a class of network consistency algorithms has been proposed to improve the efficiency of

backtracking algorithms [5, 8, 9, 10]. The idea behind the network consistency algorithms is as follows. Each individual constraint in a CSP only makes the local consistencies (consistent assignments between two variables) explicit. Through exploiting some global consistencies (consistent assignments among more than two variables), we might remove beforehand some of the domain values from consideration at each stage of a backtracking algorithm. We can use the same idea to reduce the consistency graph by removing the edges (local consistencies) that are unable to satisfy global consistencies. A consistency graph G is said to be *k-consistent* if for every $(k-1)$-clique $H = (V, E, W)$ in G, there exists at least one vertex v in every assignment set, apart from those assignment sets containing the vertices in V, such that the subgraph induced by $V \cup \{v\}$ is a k-clique. To incorporate an algorithm k-consistency, which achieves k-consistency of a graph, into the n-cliques algorithm, we can just replace the formula $\bigoplus_{j=1}^{m} Adj_G(v_{i,j})$ in the n-cliques algorithm by formula $\bigoplus_{j=1}^{m} k$-consistency$(Adj_G(v_{i,j}))$. The time complexity for the network consistency algorithms that achieve 2-consistency and 3-consistency are $O(n^2m^2)$ and $O(n^3m^5)$ respectively [8, 9].

(II). The exact expansion method tries to balance the precision of the formula in Theorem 2.3 and the efficiency of the one in Theorem 2.4. The intent is to first use Theorem 2.3 some number of times to exactly generate some subproblems, then use Theorem 2.4 to approximately solve the expanded subproblems. Each time Theorem 2.3 is used, the time complexity of the entire algorithm will increase by a factor of $O(m)$.

(III). In the n-cliques algorithm, the weight of an edge e at the end of the i^{th} iteration denotes the number of component graph sequences $A_1, \ldots, A_i$ in which the edge e occurs, where A_j is the component graph in which the edge e occurs at the j^{th} iteration. Or equivalently, it denotes the number of partial assignments to variables $x_1, \ldots, x_i$ with which the two assignments corresponding to the edge e are consistent. Due to the lack of partial assignment information, graph addition is performed without knowing to which partial assignments the weight contributes. The assignment memorization method tries to memorize the weight as well as some partial assignment information so that we can take advantage of this information and perform graph addition in a more precise way. Operationally, to memorize the most recent variable assignment, each edge of the weighted consistency graph needs to maintain an array of m weights instead of a single weight. At the end of the i^{th} iteration of the n-cliques algorithm, the j^{th} element of the weight array of an edge e, $W(e)[j]$, corresponds to the number of partial assignments to variables $x_1, \ldots, x_i$ to which e contributes with $x_i \leftarrow d_j$. Therefore, let $G_{i+1} = (V, E, W)$ be the graph at the end of the i^{th} iteration, and $Adj_{G_i}(v_{i,j}) = (V_j, E_j, W_j)$ be the j^{th} adjacency graph at the i^{th} iteration. For each edge $e \in E$, we have $W(e)[j] = \sum_{k=1}^{m} W_j(e)[k]$. The memorization of k variable assignments will cost the entire algorithm a factor of $O(m^k)$ in both time and space.

We now give the experimental performance of the algorithms described above. Experiments are performed on a set of randomly generated consistency graphs. The edges in the graphs are chosen independently and with the probability 0.75. The probability 0.75 is chosen so that the graphs have reasonably high density to have a reasonable number of cliques. Apart from the n-cliques algorithm, we also incorporate network consistency, exact expansion and assignment memorization algorithms into the n-cliques algorithm. The 2-consistency and 3-consistency algorithms achieve respectively 2-consistency and 3-consistency for each constructed adjacency graph; the 1-expansion and 2-expansion algorithms apply Theorem 2.3 once and twice respectively; the 1-memorization and 2-memorization algorithms memorize respectively the most recent one and two variable assignments. We apply each algorithm to 16 random graphs for each of the orders (4,4), (5,5), ..., (8,8).

The performance results are shown in Figure 2. Figures 2(a), 2(c) – 2(h) display the relative error of the algorithms with respect to the order of the consistency graphs. The relative error is defined as $(K - K^*)/K^*$, where K and K^* denote, respectively, the estimated and the exact number of n-cliques in the graph. The results show that the relative error of the algorithms increases as the order of the graph grows. Because the number of graph additions performed increases as the order of the graph grows, the estimation error accumulated via graph addition also increases. Furthermore, since the number of n-cliques in a graph usually grows exponentially with respect to the order of the graph, the growth rate of the relative error is also very high. However, the relation between two exact solutions usually also reflects upon the corresponding estimated solutions. This result is shown in Figure 2(b). Here we compare the relation between two exact solutions with the relation between the two corresponding estimated solutions for all pairs of the 16 random graphs. The relativity ratio is the ratio between the number of pairs that preserve the relation and the total number of pairs. The results also show that the low-order network consistency algorithms do not work well for dense graphs. In general, as the density of graphs becomes higher, the graphs are more likely to be low-order consistent. On the other hand, the exact expansion and the assignment memorization algorithms can significantly improve the estimation precision for dense graphs.

3 Transformation and Optimization Techniques

The algorithms presented in the previous section are particularly useful for programs whose constraints are explicitly expressed as a set of constraints involving comparison operators. Given the domain information about the variables in the constraints,[1] the consistency graph corresponding to the constraints can be efficiently built by using simple integer interval arithmetic and set manipulations. Given the consistency graph, nontrivial number of solutions information can be efficiently inferred as well by using the presented algorithms. This section uses an example to illustrate several program transformation and optimization techniques based on the number of solutions information, and gives experimental measurements on a set of benchmark programs. The measurements are conducted using SICStus Prolog and running on Sun4.

The benchmark programs include a crypt-arithmetic problem: the send-more-money puzzle [7]; the eight-queens problem; the five-houses puzzle [7]; a job scheduling problem [14]; a boolean satisfiability problem: a liar puzzle [7]; the magic 3×3 squares

[1] Domain information can be inferred at compile time [6] or declared by users [14].

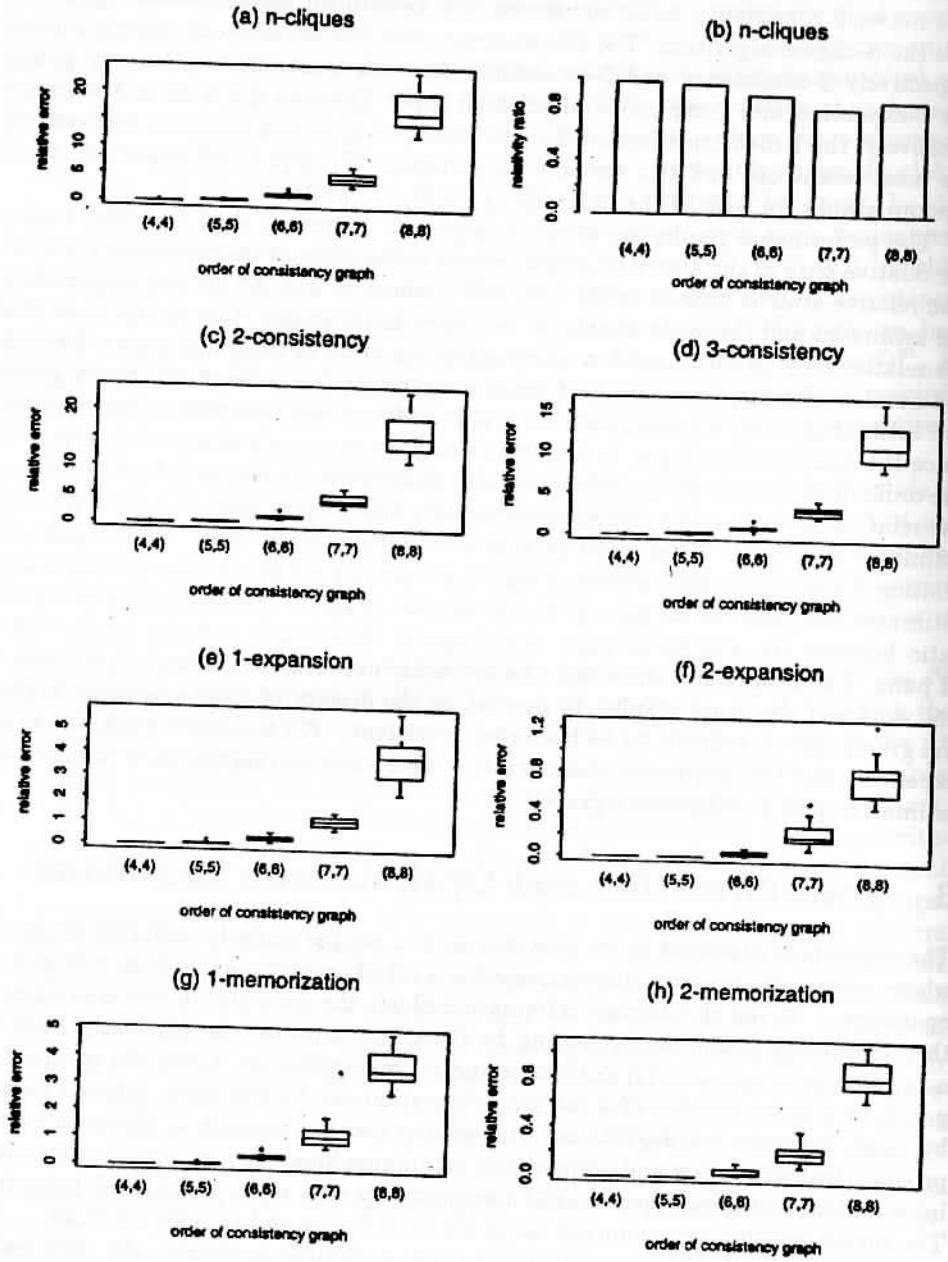


Figure 2: Experimental performance

problem [7]; a map coloring problem [1]; and a network flow problem. The example problem is a job scheduling problem [14]. The problem is to find schedules for a project that satisfy a variety of constraints among its jobs. Typical constraints in job scheduling problems include *precedence constraints*, which specify the precedence among the jobs; *disjunctive constraints*, which indicate the mutual exclusion among the jobs due to the sharing of resources; and many other constraints. An example of precedence constraints is given by the following predicate:

```
precedence_constraints(SA,SB,SC,SD,SE,SF,SG,SEnd) :-
 SB >= SA + 2, SC >= SA + 2, SD >= SA + 2, SE >= SB + 3, SE >= SC + 5,
 SF >= SD + 6, SG >= SE + 2, SG >= SF + 3, SEnd >= SG + 1.
```

Here variables `SA`,...,`SEnd` represent the starting time of jobs $A, \ldots, End$. The constraint '`SB >= SA + 2`' specifies that job A takes 2 units of time to complete and job B can be started only after job A has completed. Job End is a dummy job denoting the end of the project. An example of disjunctive constraints is given by the following predicate:

```
disj_constraints(SB,SC) :- SC >= SB + 3.
disj_constraints(SB,SC) :- SB >= SC + 5.
```

This predicate indicates that jobs B and C cannot be performed at the same time due to the sharing of resources. We can then express the constraints for the project that involves the above precedence and disjunctive constraints as follows:

```
constraints(SA,SB,SC,SD,SE,SF,SG,SEnd) :-
 prec_constraints(SA,SB,SC,SD,SE,SF,SG,SEnd), disj_constraints(SB,SC).
```

Given the duration `Du` in which the project is supposed to be carried out, a schedule that satisfies the schedule constraints can be expressed as follows:

```
schedule(Du,SA,SB,SC,SD,SE,SF,SG,SEnd) :-
  gen(Du,SA), gen(Du,SB), gen(Du,SC), gen(Du,SD), gen(Du,SE),
  gen(Du,SF), gen(Du,SG), gen(Du,SEnd),
  constraints(SA,SB,SC,SD,SE,SF,SG,SEnd).
```

Here predicate `gen/2` generates all possible time slots (integers) in a given duration.

3.1 Ordering Subgoals

Seki and Furukawa have presented a technique of transforming a generate and test program into a more efficient program by interleaving generators and constraints using the mode information of the generators [12]. In their method, subgoals are unfolded and rearranged so that constraints are interleaved into generators immediately after the constraints become active. Using this technique, the predicate `schedule/9` can be transformed into the following more efficient predicate:

Benchmarks	GT	SF	GO	SF/GO	AT
send-more-money	3600.0 ↑	34.6400	0.0781	443.53	61.329
eight-queens	2496.650	0.9879	0.9879	1.00	12.860
five-houses	3600.0 ↑	0.3451	0.1382	2.50	20.800
job-scheduling	3600.0 ↑	4.2800	0.7531	5.68	5.790
liar-puzzle	0.0659	0.0067	0.0068	0.99	0.240
magic-squares	3600.0 ↑	0.8469	0.2322	3.65	36.629
map-coloring	2608.700	3.4029	0.8948	3.80	22.160
network-flow	1595.650	0.7855	0.6649	1.18	20.439

Table 1: Measurements for goal ordering transformation (times in seconds)

```
schedule(Du,SA,SB,SC,SD,SE,SF,SG,SEnd) :-
  gen(Du,SA), gen(Du,SD), SD >= SA + 2,
  gen(Du,SF), SF >= SD + 6, gen(Du,SG), SG >= SF + 3,
  gen(Du,SE), SG >= SE + 2, gen(Du,SEnd), SEnd >= SG + 1,
  gen(Du,SB), SB >= SA + 2, SE >= SB + 3,
  gen(Du,SC), SC >= SA + 2, SE >= SC + 5, disj_constraints(SB,SC).
```

The benchmark programs are measured for a generate and test version (GT), a version using the transformation technique of Seki and Furukawa (SF), and one using our goal ordering transformation technique based on number of solutions information (GO). The result is shown in Table 1, where 3600.0↑ denotes that the corresponding execution time is over one hour. The last column AT in the table shows the analysis time for estimating the number of solutions information using the 1-expansion n-cliques algorithm. The result shows that the goal ordering transformation based on the number of solutions information produces more efficient program than Seki and Furukawa's technique for all the benchmark programs except the liar puzzle program. For this program, the 1-expansion n-cliques algorithm infers that there is only one domain value satisfying the constraints in the program for every variable. Therefore, the ordering decision is solely based on the number of forward-checkable constraints associated with each variable. However, in our simple scheme, different types of constraints are not distinguished between themselves. We can easily see that equalities usually prune more search space than inequalities, and they should have heavier weight than inequalities. The worse performance of the GO version of the liar puzzle program is mainly due to this kind of imprecision.

3.2 Reducing Variable Domains

Since the n-cliques algorithm gives an upper bound estimation, if the number of solutions associated with a variable value is inferred to be zero, we can safely remove this value from the corresponding domain at compile time. Moreover, recall that the 1-expansion n-cliques algorithm is applied to every variable to obtain the number of

Benchmarks	SF	GO	NR	DR	NR/DR	SF/DR
send-more-money	34.640	0.0781	0.0514	0.0471	1.09	777.92
five-houses	0.3451	0.1382	0.0856	0.0297	2.88	11.62
job-scheduling	4.2800	0.7531	0.4187	0.0511	8.19	83.76
liar-puzzle	0.0067	0.0068	0.0065	0.0060	1.08	1.12
network-flow	0.7855	0.6649	0.4531	0.4418	1.03	1.78

Table 2: Measurements for domain reducing optimization (times in seconds)

solutions for all the variable values. Knowing that a value cannot lead to a solution, we can also safely remove the corresponding vertex in the consistency graph without affecting correctness. This reduction can improve both the efficiency and precision of the 1-expansion n-cliques algorithm.

Therefore, for the example problem, the predicate `gen/2` can be specialized into the following new generator predicates at compile time:

```
gen_a(1).   gen_a(2).   gen_b(3).   gen_b(4).   gen_b(8).   gen_b(9).
gen_c(3).   gen_c(4).   gen_c(6).   gen_c(7).
gen_d(3).   gen_d(4).   gen_d(5).   gen_e(11).  gen_e(12).
gen_f(9).   gen_f(10).  gen_f(11).  gen_g(13).  gen_g(14).
gen_end(14). gen_end(15).
```

Experimental measurements have been conducted on the benchmark programs. Among them, the eight-queens, the magic squares and the map coloring programs have no variable values that can be removed. Since the program transformation from a generator using recursive predicate (in the version GO) to a generator using a set of facts usually also reduce the execution time, we use the version using facts to measure the sole effects from the domain reducing optimization. Combined with the goal ordering technique, the programs are executed for a version without domains reduced (NR) and one with domains reduced (DR). The results are shown in Table 2. From the measurements, combining the goal ordering transformation and the domain reducing optimization can speed up the execution of the send-more-money puzzle and the job scheduling programs by a factor of more than a hundred. Notice also that for the five-houses puzzle and the liar puzzle programs, the 1-expansion n-cliques algorithm infers that for each variable at most one value in the domain can satisfy the constraints in the problem. Consequently, after reducing the variable domains, these programs can be solved without any backtracking at runtime.

3.3 Determining Instantiation Order of Variable Values

For some applications, such as theorem proving, planning and vision problems, finding a single solution is sufficient. For some other applications, such as combinatorial optimization problems, an optimal solution is sought. In this latter situation, usually,

Benchmarks	NO	CO	NO/CO
send-more-money	22.370	9.540	2.34
eight-queens	41.590	13.530	3.07
job-scheduling	0.292	0.243	1.20
magic-squares	19.251	5.089	3.78
network-flow	0.730	0.289	2.53

Table 3: Measurements for clause ordering transformation (times in milliseconds)

the branch and bound algorithm is employed, and a first solution is sought as soon as possible so that we can start the pruning early. In all these cases, the order in which the domain values are instantiated to the variables may have a profound effect on the performance.

In the example problem, suppose we are interested in obtaining any one of the schedules that satisfy the schedule constraints rather than all of them. Then we can use the number of solutions information to determine the instantiation order of variable values. A simple heuristic is to choose the value that is associated with the most number of solutions as the first value to be instantiated. Thus using the information about the number of solutions, we can rearrange the order of the clauses in the generator predicates as follows:

```
gen_a(1).   gen_a(2).   gen_b(3).   gen_b(8).   gen_b(9).   gen_b(4).
gen_c(3).   gen_c(6).   gen_c(7).   gen_c(4).
gen_d(3).   gen_d(4).   gen_d(5).   gen_e(12).  gen_e(11).
gen_f(11).  gen_f(10).  gen_f(9).   gen_g(14).  gen_g(13).
gen_end(15). gen_end(14).
```

Experimental measurements have been conducted on the benchmark programs. Among them, the 1-expansion n-cliques algorithm infers that for the five-houses puzzle and the liar puzzle programs, there is at most one variable value satisfying the constraints in the program for each variable, and for the map coloring program, all the variable values are associated with the same number of solutions for each variable. Thus the measurements for these three programs are not included. Combined with the goal ordering technique, the programs are executed for a version using natural domain value order (NO) and a version using the order generated by our clause ordering technique (CO). The result is shown in Table 3. It shows that for all the benchmark programs, the version using clause ordering technique performs better than the version using natural order.

Acknowledgements

The author would like to thank Saumya Debray and anonymous referees for many valuable comments on the material of this paper.

References

[1] M. Bruynooghe and L. M. Pereira, "Deduction Revision by Intelligent Backtracking," *Implementation of PROLOG*, edited by J. A. Campbell, Ellis Horwood, 1984, pp. 194–215.

[2] S. A. Cook, "The Complexity of Theorem-Proving Procedures," *Conference Record 3rd Annual ACM Symposium on Theory of Computing*, 1971, pp. 151–158.

[3] R. Dechter and J. Pearl, "Network-Based Heuristics for Constraint-Satisfaction Problems," *Artificial Intelligence* 34, (1988), pp. 1–38.

[4] D. Diaz and P. Codognet, "A Minimal Extension of the WAM for clp(FD)," *Proceedings of Tenth International Conference on Logic Programming*, 1993.

[5] R. M. Haralick and G. L. Elliot, "Increasing Tree Search Efficiency for Constraint Satisfaction Problems," *Artificial Intelligence* 14, (1980), pp. 263–313.

[6] N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs," *Proceedings of ACM Symposium Principles of Programming Languages*, San Francisco, California, Jan. 1990, pp. 197–209.

[7] J. L. Lauriere, "A Language and a Program for Stating and Solving Combinatorial Problems," *Artificial Intelligence*, 10 (1), 1978, pp. 29–127.

[8] A. K. Mackworth and E. C. Freuder, "The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems," *Artificial Intelligence* 25, (1985), pp. 65–74.

[9] R. Mohr and T. C. Henderson, "Arc and Path Consistency Revisited," *Artificial Intelligence* 28, (1986), pp. 225–233.

[10] U. Montanari, "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," *Information Sciences* 7, (1974), pp. 95–132.

[11] I. Rivin and R. Zabih, "An Algebraic Approach to Constraint Satisfaction Problems," *Proceedings of Eleventh International Joint Conference on Artificial Intelligence*, August, 1989, pp. 284–289.

[12] H. Seki and K. Furukawa, "Notes on Transformation Techniques for Generate and Test Logic Programs," *Proceedings of IEEE Symposium on Logic Programming*, 1987, pp. 215–223.

[13] D. E. Smith and M. R. Genesereth, "Ordering Conjunctive Queries," *Artificial Intelligence* 26 (1985), pp. 171–215.

[14] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1989.

[15] D. H. D. Warren, "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," *Proceedings of Seventh International Conference on Very Large Data Bases*, 1981, pp. 272–281.

On the Detection of Implicit and Redundant Numeric Constraints in CLP Programs

Roberto Bagnara

Dipartimento di Informatica,
Università di Pisa,
Corso Italia 40,
56125 Pisa, Italy.
EMail: bagnara@di.unipi.it

Abstract

We address the problem of compile-time detection of implicit and redundant numeric constraints in CLP programs. We discuss how this kind of constraints have several important applications in the general field of semantics based program manipulation and, specifically, optimized compilation. We attack the problem in an original and effective way. The basic ideas underlying our approach are: (1) the use of approximate and efficient constraint inference techniques originally developed in the field of artificial intelligence. These techniques allow great flexibility in dealing with the complexity/precision tradeoff of the analysis. And (2) the integration of these techniques within an appropriate framework for the definition of non-standard semantics of CLP. We show that one notable advantage of this combined approach is that it allows to close the often existing gap between the formalization of data-flow analysis in terms of abstract interpretation and the possibility of efficient implementations. Some preliminary results (both in terms of complexity and precision) obtained with our prototype implementation are presented.

1 Introduction

Constraint logic programming (CLP) is a generalization of the pure logic programming paradigm (LP), having similar model-theoretic, fixpoint and operational semantics [14]. The CLP notion of constraint solving in a given algebraic structure encompasses the one of unification over some Herbrand universe. This gives CLP languages a great advantage, in terms of expressivity and flexibility, over LP. In some cases CLP programs can also be naturally more efficient than the correspondent LP ones, because of the ability of reasoning directly in the "domain of discourse" (e.g., real arithmetic), without requiring complicated encodings of data objects as first-order terms. However, the basic operational step in CLP program execution, a test for solvability of

constraints, is generally far more involved than unification. Furthermore, a correct implementation of a CLP language needs a *complete* solver, that is a full decision procedure for the satisfiability of constraints in the language's domain(s). The indiscriminate use of such complete solvers in their full generality can lead to severe inefficiencies. For these reasons, the optimized compilation of CLP programs can give rise to impressive performance improvements, even more impressive than the ones obtainable for the compilation of Prolog. Data-flow analysis of CLP programs has a great potential in obtaining valuable information for the compiler, and promises playing a fundamental role in the achievement of the last point in the following CLP wish list: (1) retaining the declarativity and flexibility of logic programming; (2) augmenting expressivity by allowing direct programming in the intended computational domain; and (3) gaining competitivity, in terms of efficiency, with respect to other programming paradigms. In this work we are concerned with a specific kind of analysis for CLP programs over numeric domains.

There are several CLP languages which incorporate some kind of numeric domains. Here is a list (with the particular numeric domains in parentheses) [15]: CHIP (numeric constraints over finite domains, linear rational constraints), CLP($\mathcal{R}$) (real linear arithmetic, delay mechanism for non-linear constraints), Prolog-III (linear rational arithmetic), Trilogy (linear integer arithmetic), CAL (linear rational arithmetic, possibly non-linear equations), RISC-CLP(Real) (real arithmetic), CLP(BNR) (arithmetic on real intervals), `clp(FD)` (finite domains), Echidna (finite domains and real intervals arithmetic).

Roughly speaking, the target of the data-flow analysis we present is the derivation of numeric constraints that, at some program point p, are either

implicit i.e., they are not present in the program's text at p, but they are guaranteed to hold if the computation from p has to succeed; or

truly redundant i.e., they are in the program's text at p, but they are either implied by the constraints accumulated before reaching p (in this case they could be ignored) or they will be implied by the other constraints collected through any successful computation from p (in which case they can be subject to simplified treatment). We call these constraints *past redundant* and *future redundant*, respectively[1].

redundant i.e., they are present in the program's text at p, or they are implicit at p. Notice that adding a redundant constraint c at p would result in c being future redundant.

Since in this paper we restrict ourselves to considering only clause entry and clause (successful) exit as program points, we could have used the expressions *numeric call patterns* and *numeric success patterns* to denote redundant constraints at those points. However we decided to use the implicit/redundant terminology because it helps in understanding the applications.

Our interest in automated detection of implicit and redundant numeric constraints is motivated by the wide range of applications they have in semantics-based program

[1] In [17] a more restrictive notion of future redundancy is defined.

manipulation. Moreover, while analysis techniques devoted to the discovery of implicit constraints over some Herbrand universe are well known (e.g., depth-k [22]), in the field of numeric domains very little has been done. An exception is represented by the work described in [19]. However, the analyses they propose are limited to linear constraints and use simple description domains which, though allowing for efficient implementations, cannot obtain precise information about constraints interplay.

We present a general methodology for the detection of numeric constraints which fall into the above classification. The techniques we use for reasoning about arithmetic constraints come from the world of artificial intelligence, and are known under the generic name of *constraint propagation* [8]. Notice that we do not commit ourselves to any specific CLP language, even though all the languages mentioned above can be profitably analyzed with the techniques we propose. In particular, we allow and reason about linear and non-linear constraints, integer, rational, and real numbers as well as interval domains. The work we present here was started in [2] in the restricted context of finite domains, the analysis for the detection of future redundant constraints was sketched in [4], and the constraint propagation techniques we use were presented in [5]. All the other things in this paper are new. Due to space limitations we will be very superficial on some parts, even though we try to convey all the essential ideas. However, the interested reader will find every detail in [3].

The main part of the paper is in Section 2 which describes, with some original material, the applications of redundant constraints. Section 3 gives a brief description of the constraint propagation techniques employed, while Section 4 sketches the kind of semantics treatment we have developed for describing our analysis in the framework of abstract interpretation. Some preliminary results obtained with a prototype implementation of our analyzer, even though disseminated through Section 2, are summarized in Section 5. Section 6 concludes with some remarks and directions for future work.

2 What Redundant Constraints Are For

In this section we show a number of applications for redundant constraints. Some of them are domain-independent, but we concentrate on redundant constraints over numeric domains. We will see that the range of situations where they prove to be useful is quite wide. It should then be clear that their automatic detection is very important for the whole field of semantics-based manipulation of CLP programs. The first four subsections are devoted to applications related to the compilation of CLP programs. Traditionally, this is one of the major interest areas for data-flow analysis. The remaining two subsections describe applications of redundant constraints to the improvement of other data-flow analyses.

2.1 Domain Reduction

In CLP systems supporting finite domains, like CHIP, `clp(FD)` and Echidna, variables can range over finite sets of integer numbers. These sets must be specified by the programmer. There are combinatorial problems, such as n-queens, where this

operation is trivial: variables denoting row or column indexes range over $\{1,\ldots,n\}$. For other problems, like scheduling, the ranges of variables are not so obvious. Leaving the user alone in the (tedious) task of specifying the lower and upper bounds for any variable involved in the problem is inadvisable. On one hand the user can give bounds that are too tight, thus loosing solutions. On the other hand he can exceed in being conservative by specifying bounds that are too loose. In that case he will incur inefficiency, as finite domains constraint solvers work by gradually eliminating values from variable's ranges.

A solution to this problem is either to assist the user during program development or to provide him with a compiler able to tighten the bounds he has specified. In this case the programmer can take the relaxing habit of being conservative, relying on the compiler's ability of achieving domain reduction. Whatever the programmer's habit is, domain reduction at compile-time can be an important optimization as possibly many inconsistent values can be removed once and for all from the variable's domains. This has to be contrasted with the situation where these inconsistent values are removed over and over again during the computation.

The following example is somewhat unnatural, but it shows how it can be difficult for an unassisted human to provide good variable's bounds. In contrast, it shows how relatively tight bounds can be *hidden* in a program and how they can be *discovered* by means of data-flow analysis. It is a finite domain version of the McCarthy's 91-function:

$$\begin{array}{lll} & \text{domain mc}([0,200],[0,2000]). & \\ C_1: & \text{mc}(N,M) \;:-\; & N>100, M=N-10 \;\square\,. \\ C_2: & \text{mc}(N,M) \;:-\; & N\leq 100, T=N+11 \\ & \qquad\qquad\quad \square & \text{mc}(T,U), \text{mc}(U,M). \end{array}$$

The analyzer mentioned in Section 5 is able to derive the success patterns[2] ϕ_1, for clause C_1, and ϕ_2, for clause C_2, such that:

$$\begin{array}{lll} \phi_1 & \Rightarrow & 100 < N \leq 200 \wedge 90 < M \leq 190, \\ \phi_2 & \Rightarrow & 0 \leq N \leq 100 \wedge 90 < M \leq 91 \wedge 90 < U \leq 101. \end{array}$$

Notice how the analyzer correctly infers that any finite derivation from the second clause must end up with an answer constraint entailing $M \in (90,91]$. Since variables are constrained to take integer values[3] it can be concluded that M must be bound to 91.

2.2 Extracting Determinacy

In the history of efficient Prolog execution a major role has been played by the avoidance of unnecessary backtracking, since this is the principal source of inefficiency. These efforts go back to the WAM [24] with the indexing mechanism used to reduce

[2] Every example of redundant constraint we give is one that our current prototype implementation is able to detect.

[3] Actually, this knowledge can be easily incorporated into the analyzer.

shallow backtracking. A more general way of avoiding backtracking is to use global analysis for detecting conditions under which clauses may succeed in a program (determinacy analysis). Run-time tests to check this conditions may allow for choice point elimination or, at least, for reduction of backtracking search (determinacy exploitation).

Notice that backtracking in CLP can be significantly more complex than in Prolog. The reason is that in CLP languages it is not enough to store a reference to variables that have become bound since the last choice point creation, and to unbind them on backtracking. In CLP it is generally necessary to record changes to constraints, as expressions appearing in them can assume different forms while the computation proceeds.

We show here, more or less following the exposition in [9], how redundant constraints can be used for determinacy discovery and exploitation. Consider a CLP program P and a clause in P of the form

$$C : p(\bar{X}) :- c \,\square\, q_1(\bar{X}_1), \ldots, q_n(\bar{X}_n). \tag{1}$$

Suppose now that data-flow analysis of P computes the success pattern ϕ for clause C, and the call pattern ψ for atom $p(\bar{X})$. Define the *clause condition* of C as $\Phi = \phi \wedge \psi$. Φ is a necessary condition for C to succeed when p is invoked from a successful context. In other words, every successful computation calling C is such that, on successful exit from C, the accumulated constraint entails Φ. This fact can be captured by rewriting clause C into

$$C' : p(\bar{X}) :- \Phi \wedge c \,\square\, q_1(\bar{X}_1), \ldots, q_n(\bar{X}_n). \tag{2}$$

Let now P' be the program obtained by transforming each clause of the form (1) into the form (2) as explained. It turns out that P and P' are logically equivalent, and that the exposed clause conditions can be used for detecting and exploiting determinacy in the compilation of P. In fact, suppose the predicate p being defined in P by clauses $C_1, \ldots, C_m$ with respective success patterns $\phi_1, \ldots, \phi_m$ and clause conditions $\Phi_1, \ldots, \Phi_m$. The best we can hope for is that, for each $i, j \in \{1, \ldots, m\}$ with $i \neq j$, $\phi_i \wedge \phi_j$ is unsatisfiable. In that case p is deterministic. Or, if the above condition fails, it may happen that $\Phi_i \wedge \Phi_j$ is unsatisfiable whenever $i \neq j$. Thus p might not be deterministic in itself, but we are guaranteed that in P it is always used in a determinate way. In both cases, when the conditions are simple enough to be checked, it is possible to avoid the creation of a choice point jumping directly to the single right clause. Weaker assumptions still allow to exclude clauses from search by partitioning the set of clauses into "mutually incompatible" subsets. Of course, determinacy exploitation requires the existence of an adequate indexing mechanism. As an example consider the famous CLP program expressing the Fibonacci sequence:

$$\begin{aligned}
C_1 &: \ \mathrm{fib}(N, F) \ :- \ N = 0, F = 1 \,\square\,. \\
C_2 &: \ \mathrm{fib}(N, F) \ :- \ N = 1, F = 1 \,\square\,. \\
C_3 &: \ \mathrm{fib}(N, F) \ :- \ N > 1, F = F_1 + F_2, \\
&\qquad\quad \square \ \ \mathrm{fib}(N-1, F_1), \mathrm{fib}(N-2, F_2).
\end{aligned}$$

Our analyzer derives the following success patterns[4]:

$$\begin{aligned}
\phi_1 &\Rightarrow N = 0, F = 1 \\
\phi_2 &\Rightarrow N = 1, F = 1 \\
\phi_3 &\Rightarrow N \geq 2, F \geq 2
\end{aligned}$$

Notice that, when *fib* is called with the first argument instantiated (or definite), a simple test allows to select the appropriate clause without creating a choice point. When *fib* is called with its second argument instantiated then a similar test allows at least to discriminate between $\{C_1, C_2\}$ (a choice point is necessary) and C_3 (no choice point). In both cases some calls can be made to hit an immediate failure instead of proceeding deeper before failing or looping forever, e.g., $\mathrm{fib}(1.5, X)$, $\mathrm{fib}(X, 1.5)$.

2.3 Static Call Graph Simplification

Suppose we are given a methodology for approximate deduction of implicit constraints. Then, if the approximate constraint system has a non-trivial notion of consistency[5], we also have a methodology for approximate consistency checking which we can use for control-flow analysis of CLP programs. By soundness, when *false* is derived as an implicit constraint we can safely conclude that the original set of constraints was unsatisfiable, and that the computation branch responsible for this state of affairs is dead, i.e., it cannot possibly lead to any success.

This information can be employed at compile-time to generate a simplified call graph for the program at hand. Let

$$p(\bar{X}) :- q_1(\bar{Y}_1), \ldots, q_n(\bar{Y}_n).$$

be a program clause, and let the predicate q_i, $1 \leq i \leq n$, be defined by clauses $C_{i1}, \ldots, C_{im}$. While performing the analysis we may discover that whenever we use clause C_{ij}, with $1 \leq j \leq m$, to resolve with $q_i(\bar{Y}_i)$, we end up with an unsatisfiable constraint. In this case we can drop the edge from the $q_i(\bar{Y}_i)$ call in the above clause to C_{ij} from the syntactic call graph of the program. This simplification can be used for generating faster code[6]. We illustrate this point by means of an example. Our analyzer produces the following call graph representation, when presented with the *fib* program:

$$C_3 \ :- \ \{C_2, C_3\}, \{C_1, C_2, C_3\}.$$

The above notation can be read as follows: if a call to clause C_3 has to succeed, then the first atom will give rise to a call whose range is restricted to clauses C_2 and C_3, while the second atom will result in an unrestricted call, i.e., whose range is constituted by C_1, C_2, and C_3. In other words, when treating the first recursive

[4] The same patterns would have been derived even if $N > 1$ did not appear in C_3.

[5] This is not the case for, let's say, groundness or definiteness analysis, where constraints are of the form $X = gnd$ and $X = any$. It is our case where, clearly, $\{X < 0, X \geq 1\} \vdash false$ (see Section 3), and the case of depth-k abstraction [22].

[6] We do not want to make a strong claim here, but we have not found any previously published proposal for this optimization.

call of clause C_3, clause C_1 can be forgotten. This information allows for search space reduction without any overhead, in the case where the third clause of *fib* is called with the first argument uninstantiated, that is, when search is not avoidable. Figure 1 shows how this can be achieved by means of a simple compilation scheme in the setting of the WAM and its extensions [16].

```
fib/3_1:    try_me_else fib/3_2
            < code for clause 1 >
fib/3_2:    retry_me_else fib/3_3
fib/3_2a:   < code for clause 2 >
fib/3_3:    trust_me
fib/3_3a:   ...
            call fib/3_2_3
            call fib/3_1
            ...

fib/3_2_3:  try fib/3_2a
            trust fib/3_3a
```

Figure 1: Call graph simplification: fragment of WAM-like code for the 3rd clause of *fib* to be executed when the first argument is not definite.

Notice how this simple transformation reduces the amount of backtracking. In fact every time clause C_3 is invoked a pointless call to clause C_1 is avoided, with the consequent saving of one backtracking. Indeed, it is easy to think about more involved examples where the pruned computation branch would have proceeded deeper before failing, thus wasting more work. When the number of applicable clauses is found to be one, a choice point can be avoided, thus achieving greater savings. In these cases determinacy is exploited without any run-time effort. In contrast, the optimization is always achieved without any time overhead at the price of at most a small, constant increase of space usage for additional code. An example where choice point creations are avoided is the following:

$$
\begin{array}{lll}
C_1: & \text{square}(N, S_N) & :- \quad N = 0, S_N = 0 \ \square\,. \\
C_2: & \text{square}(N, S_N) & :- \quad N = M + 1, S_N = S_M + 2 * M + 1 \\
 & & \quad \square \ \text{square}(M, S_M).
\end{array}
$$

$$
\begin{array}{lll}
C_3: & \text{square3}(X, Y, Z) & :- \quad X < Y, Y < Z, S_X + S_Y = S_Z \\
 & & \quad \square \ \text{square}(X, S_X), \text{square}(Y, S_Y), \\
 & & \quad\quad \text{square}(Z, S_Z).
\end{array}
$$

where the detected call graph is given by

$C_2 :- \{C_1, C_2\}.$ $\qquad$ $C_3 :- \{C_1, C_2\}, \{C_2\}, \{C_2\}.$

2.4 Future Redundant Constraints Optimization

As mentioned in the introduction, we say that a constraint is *future redundant* if, after the satisfiability check, adding or not adding it to the *current constraint* (i.e., the constraint accumulated so far in the computation), will not affect any answer constraints. Consider the (by now standard) example:

$$
\begin{array}{lll}
C_1: & \text{mortgage}(P, T, I, R, B) & :- \quad T = 1, \\
 & & \quad B = P * (1 + I/1200) - R \ \square\,. \\
C_2: & \text{mortgage}(P, T, I, R, B) & :- \quad T > 1, T_1 = T - 1, P \geq 0, \\
 & & \quad P_1 = P * (1 + I/1200) - R \\
 & & \quad \square \ \textit{mortgage}(P_1, T_1, I, R, B).
\end{array}
$$

In any derivation from the second clause the constraint $T_1 = T - 1 \wedge T_1 > 1$ or $T_1 = T - 1 \wedge T_1 = 1$ will be encountered, and both imply $T > 1$. Thus $T > 1$ in the second clause is future redundant. If T is uninstantiated, not adding the future redundant constraint reduces the "size" of the current constraint, thus reducing the complexity of any subsequent satisfiability check. A dramatic speed-up is obtainable thanks to this optimization [17]. Notice that the definition of future redundant constraint given in [17] is more restrictive than ours and, while allowing a stronger result for the equivalence of the optimized program with respect to the original one, it fails to capture situations which can be important not only for compilation purposes (see Section 2.5.1 below). As an example, consider a version of *fib* having a constraint $F \geq 2$ or weaker in the recursive clause. This is future redundant for our definition (and is recognized by the analyzer), while it is not such for the definition in [17].

2.5 Improving any Other Analysis

It is well known that different data-flow analyses can be combined together. In the framework of abstract interpretation this is achieved by means of standard constructions such as direct reduced product [7]. The combined analysis can be more precise than each of the component ones. However, what we want to emphasize here, is that redundant constraint analyses can improve both the precision and the efficiency of any other analysis. This is due to the ability, described in Section 2.3, of discovering computation branches which are dead. The obvious implication is that these branches can be safely excluded from analysis, thus resulting in improved efficiency, because less branches need to be analyzed, and better precision, because the *merge-over-all-paths* operation needed for ensuring soundness [6] has potentially a less dramatic effect. We illustrate this last point by means of an example. Consider the following predicate definition:

$$
\begin{array}{lll}
C_1: & \text{r}(X, Y, Z) & :- \quad Y < X, Z = 0 \ \square\,. \\
C_2: & \text{r}(X, Y, Z) & :- \quad Y \geq X, Z = Y - X \ \square\,.
\end{array}
$$

This defines the so called *ramp function*, and is one of the linear piecewise functions which are used to build simple mathematical models of valuing options and other

financial instruments such as stocks and bonds [18]. Suppose we are interested in definiteness analysis[7] of a program containing the above clauses. A standard definiteness analyzer cannot derive any useful exit pattern for a call to $r(X,Y,Z)$ whose context is $X = any \wedge Y = any \wedge Z = any$, even though the "real" call pattern implied $Y < X$. In contrast, it may well be the case that, in the same situation, the definiteness analyzer combined with ours (or supplemented with the simplified call graph described in Section 2.3) can deduce the exit pattern $Z = gnd$. This is due to the ability of recognizing that, in the mentioned context, only clause C_1 is applicable. Indeed, this is what happens in the *option* program distributed with the CLP($\mathcal{R}$) system.

2.5.1 Improving the Results of some Other Analyses

The previous section showed how our analysis can generally improve the others. There are, however, more specific situations where its results may be of help. For example, the freeness analysis proposed in [10] can be greatly improved by the detection of future redundant constraints. In fact, their abstraction is such that constraints like $N \geq 0$ "destroy" (sometimes unnecessarily) the freeness of N. This kind of constraints are very commonly used as clause guards, and many of them can be recognized as being future redundant. This information imply that they do not need to be abstracted, with the corresponding precision gain.

The analysis described in [12] aims at the compile-time detection of those non-linear constraints, which are delayed in the CLP($\mathcal{R}$) implementation, that will become linear at run time. This analysis is important for remedying the limitation of CLP($\mathcal{R}$) to linear constraints by incorporating powerful (and computationally complex) methods from computer algebra as the ones employed in RISC-CLP(Real). With the results of the above analysis this extension can be done in a smooth way: non-linear constraints which are guaranteed to become linear will be simply delayed, while only the other non-linear constraints will be treated with the special solving techniques. Thus, programs not requiring the extra power of these techniques will be hopefully recognized as such, and will not pay any penalties. The analysis of [12] is a kind of definiteness. One of its difficulties shows up when considering the simplest non-linear constraint: $X = Y * Z$. Clearly X is definite if Y and Z are such. But we cannot conclude that the definiteness of Y follows from the one of X and Z, as we need also the condition $Z \neq 0$. Similarly, we would like to conclude that X is definite if Y or Z have a zero value. It should then be clear how the results of the analysis we propose can be of help: by providing approximations of the concrete values of variables, something which is not captured by common definiteness analyses while being crucial when dealing with non-linear constraints.

3 Arithmetic Reasoning

Our analysis is based on constraint inference (a variant of constraint propagation) [8]. This technique, developed in the field of AI, has been applied to temporal and spatial reasoning [1, 23].

[7] I.e. aimed at discovering variables which are constrained to a unique value.

Let us focus our attention to arithmetic domains, where the constraints are binary relations over expressions. We represent them by means of labelled digraphs. Nodes are called *quantities* and are labeled with the corresponding arithmetic expression and a variation interval. Edges are labelled with relation symbols. An example appears in figure 2. Conjunction of constraints is handled by connecting digraphs in the obvious way, merging the nodes having equal labels.

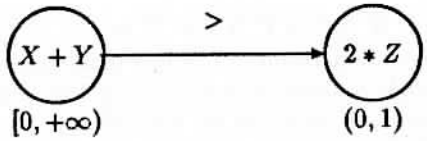


Figure 2: Simple example of constraint network representing the (conjunctive) set of constraints $\{X + Y > 2 * Z, X + Y \in [0, +\infty), 2 * Z \in (0, 1)\}$.

Let us consider a digraph representing a conjunction of constraints. We can *enrich* it by either adding new edges to it or by adding (stronger) relations to the labels of existing edges or by refining the interval labels. Of course, we are interested in approximate but *sound* deduction. To this purpose we use a number of computationally efficient techniques, both qualitative (on the ordinal relationships between arithmetic expressions) and quantitative (on the values these expressions can take). All these techniques are integrated, that is, inference made with one technique can trigger further inferences by the other ones. As a result the range of arithmetic inferences the system is able to perform is quite wide and suitable for our application. An example of qualitative technique is computing the *transitive closure* of the constraint digraph using the following table:

$\parallel$	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$
$<$	$<$	$<$	??	??	$<$	??
$\leq$	$<$	$\leq$	??	??	$\leq$	??
$>$	??	??	$>$	$>$	$>$	??
$\geq$	??	??	$>$	$\geq$	$\geq$	??
$=$	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$
$\neq$	??	??	??	??	$\neq$	??

New relationships are found by looking up the relevant table entries. For example, the system infers $A < C$ from $A \leq B$, $B < C$. A classical quantitative technique is *interval arithmetic* which allows to infer the variation interval of an expression from the intervals of its sub-expressions. An example inference is: $A \in [3, 6) \wedge B \in [-1, 5] \vdash A + B \in [2, 11)$. Another important technique is *relational arithmetic* [23] which infers constraints on the qualitative relationship of an expression to its arguments. It is encoded by a number of axiom schema, for example, for each $\bowtie \in \{=, \neq, \leq <, \geq, >\}$:

$$
\begin{array}{rcl}
x \bowtie 0 & \Rightarrow & (x+y) \bowtie y \\
(x>0 \wedge y>0) & \Rightarrow & \left\{ \begin{array}{l} x \bowtie 1 \Rightarrow (x*y) \bowtie y \\ y \bowtie 1 \Rightarrow (x*y) \bowtie x \end{array} \right. \\
(x>0 \wedge y<0) & \Rightarrow & (x \bowtie -y \Rightarrow -1 \bowtie (x/y)) \\
x \bowtie y & \Rightarrow & e^x \bowtie e^y
\end{array}
$$

An example of inference is: $X \geq 0 \wedge Y \geq 0 \vdash X+1 \leq Y+2X+1$. Notice that there is no restriction to linear constraints. The last technique we mention is *numeric constraint propagation*, which consists in determining the relationship between two quantities when their associated intervals do not overlap, except possibly at their endpoints. For example, if $A \in (-\infty, 2]$, $B \in [2, +\infty)$, and $C \in [5, 10]$, we can infer that $A \leq B$ and $A < C$. It is also possible to go the other way around, i.e., knowing that $U < V$ may allow to refine the intervals associated to U and V so that they do not overlap. The integration of these techniques, and possibly others, allows to obtain a very good tradeoff between inferential power and computational complexity. We refer to [3] for an extensive treatment of these issues.

4 Generalized, Concrete, and Abstract Semantics

We adopt a generalized semantics approach, i.e. where semantics is parameterized with respect to an underlying constraint system, as in [11]. The main advantages are that: (1) different instances of CLP can be used to define non-standard semantics for constraint logic programs; and (2) the abstract interpretation of CLP programs can be thus formalized inside the CLP paradigm. To achieve this last point one has to define an "abstract" (or, in our case, approximate) constraint system, which soundly captures the interesting properties of the "concrete" one. Then the abstract and concrete semantics are more easily related, being instances (over two different constraint systems) of the same generalized semantics.

The constraint systems we use in our formulation are built starting from the ones defined in [21]. They are constituted by a set $\mathcal{C}$ of atomic constraints (e.g., $X*Y = 2*Z$) and an *entailment* relation $\vdash$ over the subsets of $\mathcal{C}$ satisfying some very reasonable conditions (reflexivity and transitivity). In the concrete semantics the entailment relation $\vdash^{\flat}$ is defined by the logical theory axiomatizing the computation domain (e.g., the theory RCF of real closed fields). In the approximate semantics the entailment $\vdash^{\natural}$ will be "less powerful" (e.g., the one defined by the inference techniques of Section 3). Of course, we must ensure soundness which, roughly speaking, amounts to saying that $C' \vdash^{\natural} C'' \Rightarrow C' \vdash^{\flat} C''$. Any entailment relation defines an *upper closure operator* over $\wp(\mathcal{C})$: $\rho(C) = \{ c \in \mathcal{C} \mid C \vdash c \}$. Then a minimal first-order structure is built over the above constraint systems by introducing (soundly correlated concrete and abstract versions of) *hiding operators* $\exists_X$, modelling the projection of constraints over variables, and *diagonal elements* d_{XY}, modelling "parameter passing". Our generalized semantics of a CLP program P is constructed from these building blocks.

Here is the immediate consequence operator defining the bottom-up version:

$$
T_P(I) = \bigoplus_{C \in P} \left\{ p(\bar{X}) :- \exists_{\bar{X}} \tilde{c} \;\middle|\; \begin{array}{l} C : p(\bar{X}) :- c \,\square\, p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n); \\ vars(C) \cap vars(I) = \emptyset \\ \forall i = 1, \ldots, n : \\ \quad p_i(\bar{Y}_i) :- c_i \in I,\ c_i' = d_{\bar{X}_i, \bar{Y}_i} \otimes c_i; \\ \tilde{c} = c \otimes c_1' \otimes \cdots \otimes c_n'; \\ \tilde{c} \not\vdash false. \end{array} \right\},
$$

where $C_1 \otimes C_2 = \rho(C_1 \cup C_2)$ models (concrete and approximate) conjunction of constraints, while two different flavors of $\oplus$ are used to capture disjunction in the concrete semantics (where the constraint arising from all the computation paths must be kept distinct) and in the approximate one (where these constraints are merged together). As a concluding remark, notice that, in order to ensure the finiteness of our analysis, a widening/narrowing approach [6] is used in the abstract fixpoint computation. This is due to the presence of unbounded intervals in our approximate constraint system. The interested reader will find all the details in [3].

5 Preliminary Results

Our work on numeric redundant constraint analysis has been concretized in a prototype analyzer based on the ideas sketched in Sections 3 and 4. During the design and implementation of the prototype we had two objectives in mind: (1) to demonstrate the feasibility of the approach; and (2) to retain as much "declarativity" as possible, in order to end up with an executable specification which would be very useful for developing more refined and efficient implementations. The result of this work is constituted by about 3200 lines of SICStus Prolog (98% portable). The only syntax currently accepted by the analyzer is the one of CLP($\mathcal{R}$). We plan to accommodate at least CHIP and Prolog-III in future versions. We cannot describe here the implementation, so we just give some preliminary results of its use. Table 1 reports, for each of five different programs, the analysis time (a Sun SPARCstation 10 was used) and a description of the benefits obtained. We have met the first four programs before; the last one, *option*, is the more complex and is distributed with the CLP($\mathcal{R}$) system.

Notice that the current prototype does not make use of sophisticated fixpoint computation techniques. Furthermore, we have not exploited yet one of the big advantages of constraint propagation techniques: incrementality. For example, even though we compile relational and interval arithmetic down to quite efficient code, we "execute" the entire code sequence irrespectively of whether the constraint network contains enough additional information to obtain something new. In a more refined implementation we would use the generated code to attach *demons* to quantities. These demons would be fired only on the occurrence of relevant changes in the network. Similar considerations can be done for the numeric constraint propagation technique. Even bigger is the room for improvements of transitive closure (now absorbing 60% of the prototype's execution time). First of all we are currently using a naive variation of Warshall/Warren algorithm for graph closure [26, 25], since it

Program	Analysis Time	Benefits
fib	1640 ms	• 1 future redundant constraints ($N > 1$); • deterministic if 1st argument definite[a]; • partially deterministic if 2nd argument definite; • 1 simplified procedure call. [a]The success pattern $N \geq 2$ would have been found even in case $N > 1$ was not present in the recursive clause.
mc91	440 ms	• domain reduction[a]; • partially deterministic if 2nd argument definite[b]. [a]This is the only program for which we have have given a finite domain version. Of course, domain reduction would be achieved for other programs too. [b]It is also deterministic if the 1st argument is definite.
square3	1180 ms	• deterministic if 1st argument definite; • deterministic if 2nd argument definite; • 2 simplified (deterministic) procedure calls.
mortgage	200 ms	• 1 future redundant constraint ($N > 1$).
option	11440 ms	• 2 simplified (deterministic) procedure calls.

Table 1: Some results obtained by the prototype analyzer implementation.

is more suitable for the simple Prolog data structures we employ. Much more efficient techniques exist for this task which make use of particular information about the graph structure [13]. Secondly, we do not currently make use of the constraint network's "macro-structure". The networks arising from our analysis method can be partitioned into sub-networks being connected in a very simple tree structure. The root is constituted by the network corresponding to a program clause, the leaves are networks coming from constrained atoms in the current interpretation, one for each atom in the clause body. The root is connected to its children through sheaves of edges (i.e., equality constraints). This is a restricted kind of a structure defined and studied in [20]. Roughly speaking, this allows transitive closure to be applied on a local sub-network basis, and to become global only if new edges (constraints) are added between the nodes (quantities) making up the "interfaces" between sub-networks.

In summary, many optimizations and clever programming techniques are possible, and we expect the new analyzer's implementation we are designing will attain a performance improvement of one to two orders of magnitude over the current prototype.

6 Conclusions

We have shown that the compile-time detection of implicit and redundant numeric constraints in CLP programs can play a crucial role in the field of semantics based program manipulation. This is especially true for the area of optimized compilation,

where they can enable major performance leaps. Nonetheless, and despite the fact that almost every existing CLP language incorporates some kind of numeric domain, this research topic is relatively unexplored. We have attacked the problem by adapting efficient reasoning techniques originating from the world of artificial intelligence, where approximate deduction holds the spotlight since the origins. These techniques have then be integrated into a generalized semantics framework, allowing the easy formalization of our data-flow analysis in terms of abstract interpretation. The preliminary results obtained with our prototype implementation justify our belief that we are on the right way towards satisfactory solutions for the problems of data-flow analysis and highly optimized compilation of CLP programs.

Current and future work includes: (1) studying variants of depth-k approximations to be integrated with ours[8] for more precise analysis of *real* programs; (2) development of an "highly-engineered" version of the approximate constraint solver; and (3) study of the possible extensions of this work to the concurrent constraint programming paradigm.

References

[1] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] R. Bagnara. Interpretazione Astratta di Linguaggi Logici con Vincoli su Domini Finiti. M.Sc. dissertation, University of Pisa, July 1992. In Italian.

[3] R. Bagnara. Determination of Redundant Constraints in CLP Languages: Theory and Application. Technical report, Dipartimento di Informatica, Università di Pisa, 1994. Forthcoming.

[4] R. Bagnara, R. Giacobazzi, and G. Levi. Static Analysis of CLP Programs over Numeric Domains. In M. Billaud, P. Castéran, MM. Corsini, K. Musumbu, and A. Rauzy, editors, *Actes "Workshop on Static Analysis '92"*, volume 81–82 of *Bigre*, pages 43–50, Bordeaux, September 1992. Extended abstract.

[5] R. Bagnara, R. Giacobazzi, and G. Levi. An Application of Constraint Propagation to Data-Flow Analysis. In *Proceedings of "The Ninth Conference on Artificial Intelligence for Applications"*, pages 270–276, Orlando, Florida, March 1993. IEEE Computer Society Press, Los Alamitos, CA.

[6] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

[7] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.

[8] E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, 32:281–331, 1987.

[9] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Extracting Determinacy in Logic Programs. In *Tenth International Conference on Logic Programming*, pages 100–115. MIT Press, June 1993.

[8] Standard depth-k is already part of the analysis, something we could not talk about in this paper.