# 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:

$$\begin{array}{llll} & \multicolumn{3}{l}{\text{domain mc}\big([0,200],[0,2000]\big).} \\ C_1: & \mathrm{mc}(N,M) & :- & N > 100, M = N - 10 \,\Box\, . \\ C_2: & \mathrm{mc}(N,M) & :- & N \leq 100, T = N + 11 \\ & & \Box & \mathrm{mc}(T,U), \mathrm{mc}(U,M). \end{array}$$

The analyzer mentioned in Section 5 is able to derive the success patterns[2] $\phi_1$, for clause $C_1$, and $\phi_2$, 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

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

Suppose now that data-flow analysis of $P$ computes the success pattern $\phi$ for clause $C$, and the call pattern $\psi$ for atom $p(\bar{X})$. Define the *clause condition* of $C$ as $\Phi = \phi \wedge \psi$. $\Phi$ 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 $\Phi$. This fact can be captured by rewriting clause $C$ into

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

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:

$$\begin{array}{llll}
C_1 : & \mathrm{fib}(N, F) & :\!- & N = 0, F = 1 \,\Box\, . \\
C_2 : & \mathrm{fib}(N, F) & :\!- & N = 1, F = 1 \,\Box\, . \\
C_3 : & \mathrm{fib}(N, F) & :\!- & N > 1, F = F_1 + F_2, \\
 & & \Box & \mathrm{fib}(N-1, F_1), \mathrm{fib}(N-2, F_2).
\end{array}$$

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 $C_{i1}$, ..., $C_{im}$. 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:

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

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 $X = \mathit{gnd}$ and $X = \mathit{any}$. It is our case where, clearly, $\{X < 0, X \geq 1\} \vdash \mathit{false}$ (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:

$$
\begin{array}{llcl}
C_1: & \text{square}(N, S_N) & :- & N = 0, S_N = 0 \,\square\, . \\
C_2: & \text{square}(N, S_N) & :- & N = M + 1, S_N = S_M + 2 * M + 1 \\
 & & \square & \text{square}(M, S_M). \\
 & & & \\
C_3: & \text{square3}(X, Y, Z) & :- & X < Y, Y < Z, S_X + S_Y = S_Z \\
 & & \square & \text{square}(X, S_X), \text{square}(Y, S_Y), \\
 & & & \text{square}(Z, S_Z).
\end{array}
$$

where the detected call graph is given by

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

### 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:

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

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:

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

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.

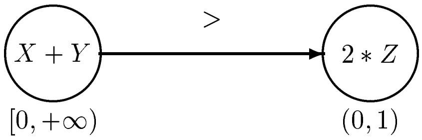


Figure 2: Simple example of constraint network representing the (conjunctive) set of constraints $\{X+Y > 2*Z, X+Y \in [0,+\infty), 2*Z \in (0,1)\}$.

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:

| | $<$ | $\leq$ | $>$ | $\geq$ | $=$ | $\neq$ |
|---|---|---|---|---|---|---|
| $<$ | $<$ | $<$ | ?? | ?? | $<$ | ?? |
| $\leq$ | $<$ | $\leq$ | ?? | ?? | $\leq$ | ?? |
| $>$ | ?? | ?? | $>$ | $>$ | $>$ | ?? |
| $\geq$ | ?? | ?? | $>$ | $\geq$ | $\geq$ | ?? |
| $=$ | $<$ | $\leq$ | $>$ | $\geq$ | $=$ | $\neq$ |
| $\neq$ | ?? | ?? | ?? | ?? | $\neq$ | ?? |

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 $\bowtie \in \{=, \neq, \leq <, \geq, >\}$:

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

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 $\vdash^\sharp$ 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 $C' \vdash^\sharp C'' \Rightarrow C' \vdash^\flat C''$. Any entailment relation defines an *upper closure operator* over $\wp(\mathcal{C})$: $\rho(C) = \bigl\{\, c \in \mathcal{C} \bigm| C \vdash c \,\bigr\}$. 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:

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

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 ($N > 1$);<br>• deterministic if 1st argument definite[a];<br>• partially deterministic if 2nd argument definite;<br>• 1 simplified procedure call.<br><br>[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];<br>• partially deterministic if 2nd argument definite[b].<br><br>[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.<br>[b] It is also deterministic if the 1st argument is definite. |
| *square3* | 1180 ms | • deterministic if 1st argument definite;<br>• deterministic if 2nd argument definite;<br>• 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.

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

[11] R. Giacobazzi, S. K. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 581–591, 1992.

[12] Michael Hanus. Analysis of nonlinear constraints in CLP($\mathcal{R}$). In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Progra mming*, pages 83–99, Budapest, Hungary, 1993. The MIT Press.

[13] Y. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive Closure Algorithms Based on Graph Traversal. *ACM Transactions on Database Systems*, 18(3):512–576, September 1993.

[14] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[15] J. Jaffar and M. Maher. Constraint Logic Programming: a Survey. To appear in the Journal of Logic Programming, 1994.

[16] J. Jaffar, S. Michaylov, P. Stuckey, and R.H.C Yap. An Abstract Machine for CLP($\mathcal{R}$). In *ACM Programming Language Design and Implementation*, volume 27 of *SIGPLAN Notices*, pages 128–139. ACM Press, 1992.

[17] N. Jørgensen, K. Marriot, and S. Michaylov. Some Global Compile-Time Optimizations for CLP($\mathcal{R}$). In *Proc. 1991 Int'l Symposium on Logic Programming*, pages 420–434, 1991.

[18] C. Lassez, K. McAloon, and R. Yap. Constraint Logic Programming and Option Trading. *IEEE Expert*, 2(3), 1987.

[19] K. Marriott and P. Stuckey. The 3 R's of optimizing Constraint Logic Programs: Refinement, Removal and Reordering. In *Proc. Twentieth Annual ACM Symp. on Principles of Programming Languages*, pages 334–344. ACM Press, 1993.

[20] U. Montanari and F. Rossi. Constraint Relaxation may be Perfect. *Artificial Intelligence Journal*, 48:143–170, 1991.

[21] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundation of Concurrent Constraint Programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, pages 333–353. ACM, 1991.

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

[23] R. Simmons. Commonsense Arithmetic Reasoning. In *Proc. AAAI-86*, pages 118–124, 1986.

[24] D. H. Warren. An Abstract Prolog Instruction Set. Technical Report Note 309, SRI International, 1983.

[25] H. S. Warren. A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations. *Communications of the ACM*, 18(4):218–220, April 1975.

[26] S. Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11–12, January 1962.