In *Proc. of the Tenth International Conference on Logic Programming*,
Budapest, June 1993, pp. 83–99, MIT Press

Analysis of Nonlinear Constraints in CLP($\mathcal{R}$)

Michael Hanus
Max-Planck-Institut für Informatik
Im Stadtwald, W-6600 Saarbrücken, Germany
michael@mpi-sb.mpg.de

Abstract

Solving nonlinear constraints over real numbers is a complex problem. Hence constraint logic programming languages like CLP($\mathcal{R}$) or Prolog III solve only linear constraints and delay nonlinear constraints until they become linear. This efficient implementation method has the disadvantage that sometimes computed answers are unsatisfiable or infinite loops occur due to the unsatisfiability of delayed nonlinear constraints. These problems could be solved by using a more powerful constraint solver which can deal with nonlinear constraints like in RISC-CLP(Real). Since such powerful constraint solvers are not very efficient, we propose a compromise between these two extremes. We characterize a class of CLP($\mathcal{R}$) programs for which all delayed nonlinear constraints become linear at run time. Programs belonging to this class can be safely executed with the efficient CLP($\mathcal{R}$) method while the remaining programs need a more powerful constraint solver.

1 Introduction

The constraint logic programming paradigm [13] generalizes logic programming by replacing the Herbrand universe of terms by other, in general more powerful, domains. Unification of terms is replaced by solving constraints over these domains. For instance, CLP($\mathcal{R}$) [15, 11] adds real numbers to the Herbrand universe and contains equations and inequations as constraints. The system includes a constraint solver over the real numbers. Since solving nonlinear constraints is a complex problem, the constraint solver in CLP($\mathcal{R}$) is restricted to linear constraints. Nonlinear constraints are delayed until some variables in these constraints get unique values during the further computation process so that the delayed constraints become linear [16] (this approach is also taken in Prolog III [4]). If a computation stops with some delayed nonlinear constraints, the system generates a "maybe" answer, i.e., it is not ensured that a solution exists.

Example 1 Consider the following CLP($\mathcal{R}$) program to compute mortgage payments:

```
mortgage(P,T,IR,B,MP) :-
        T > 0,  T <= 1,  B = P*(1+T*IR) - T*MP.
mortgage(P,T,IR,B,MP) :-
        T > 1,  mortgage(P*(1+IR)-MP, T-1, IR, B, MP).
```

The parameters are the principal `P`, the life of the mortgage `T` (in months), the monthly interest rate `IR`, the outstanding balance `B`, and the monthly payment `MP`. Due to the constraint solving mechanism this program can be queried in different ways. The query

```
?- mortgage(100000, T, 0.01, 0, 1400).
```

asks for the time to finance a mortgage, and the answer constraint is `T=125.901`. The query

```
?- mortgage(P, 180, 0.01, B, MP).
```

asks for a relationship between the principal, the outstanding balance and the monthly payment, and the answer constraint is `P=0.166783*B+83.3217*MP`. But if we want to compute the interest rate as in the query

```
?- mortgage(1000, 2, IR, 0, 600).
```

CLP($\mathcal{R}$) cannot compute a solved answer due to the restriction to linear constraints. The computed answer is `600=(1000*IR+400)*(IR+1)`. □

The CLP($\mathcal{R}$) method of delaying nonlinear constraints and solving only linear constraints is efficient and successful for many applications. However, there are also programs where this method is not sufficient because CLP($\mathcal{R}$) continues a computation with unsatisfiable nonlinear constraints. This may generate unsatisfiable answers or infinite loops. Such problems can be avoided if a more powerful constraint solver is used. For instance, CAL [2] and RISC-CLP(Real) [12] do not delay nonlinear constraints but apply special methods from computer algebra to check the satisfiability of all constraints.

Example 2 [12] Consider the following program for computing Pythagorean numbers:

```
nat(X) :- X = 1.
nat(X) :- X > 1, nat(X-1).
pyth(X,Y,Z) :- X*X+Y*Y = Z*Z, X <= Y, nat(Z), nat(X), nat(Y).
```

CLP($\mathcal{R}$) runs into an infinite loop for the query `?-pyth(X,Y,Z)` since it does not detect the unsatisfiability of the nonlinear constraints, while RISC-CLP(Real) computes the answers `X=3,Y=4,Z=5`, `X=6,Y=8,Z=10`, etc. □

Unfortunately, it is difficult to deal with nonlinear constraints, and constraint solvers for nonlinear constraints are not very efficient. It is undesirable to use such complex constraint solvers for problems which can be solved by the CLP($\mathcal{R}$) method. Therefore we propose a compromise between these two extremes. In the following we will characterize a class of CLP($\mathcal{R}$) programs for which all delayed nonlinear constraints become linear at run time. Since such a property is undecidable in general, our characterization is based on a compile-time analysis of CLP($\mathcal{R}$) programs using abstract interpretation techniques. Consequently, we cannot give a precise characterization of this class of programs but we compute a safe approximation of it. It is ensured that the CLP($\mathcal{R}$) computation of a program belonging to this approximated class does not stop with delayed nonlinear constraints.

Our method analyses the nonlinear constraints which may occur at run time. A *nonlinear constraint* is an equation or inequation containing an expression `X*Y` where both `X` and `Y` do not have unique values. In order to decide whether such a constraint becomes linear, we must know if `X` or `Y` are constrained to unique values. Thus we need a program analysis corresponding to groundness analysis in logic programming [3, 7, 22]. A groundness analysis where variables are simply abstracted into values like *ground*, *free* or *any* is not sufficient for our purpose since in constraint logic programming variables often become ground due to the addition of new constraints. For instance, consider the following sequence of constraints:

```
?- Z = X*Y,  X = A+B,  C = 3+A,  B = 5,  C = 6.
```

A simple groundness analysis would infer that only `B` and `C` are ground after the left-to-right evaluation of this goal. But due to the constraint solving mechanism, also `A` and `X` become ground and therefore the constraint `Z=X*Y` is linear at the end of this goal.[1] In order to provide an analysis of such situations, our method considers the dependencies of variables in constraints and approximates the grounding of variables due to constraint solving.

In the next section we give a short description of the syntax and the operational semantics of a restricted class of CLP($\mathcal{R}$) programs for which our analysis is designed. The abstract domain and the abstract interpretation algorithm for the analysis of nonlinear constraints in CLP($\mathcal{R}$) programs are presented in Section 3. The correctness of our method is outlined in Section 4. In Section 5 we show the extension of our method to other delayed constraints which may occur in CLP($\mathcal{R}$) programs. Finally, we discuss possible applications of our method in Section 6.

2 Operational semantics of CLP($\mathcal{R}$) programs

In this section we present the class of CLP($\mathcal{R}$) programs which we will analyse together with their operational semantics.

A CLP($\mathcal{R}$) program is a collection of Horn clauses where some functors and predicates have a predefined meaning. *Terms* are built from variables, numeric constants (real numbers), atoms (string constants), uninterpreted functor symbols with a positive arity, and the predefined arithmetic functions `+`, `-` and `*`.[2] An *arithmetic term* does not contain atoms and uninterpreted functor symbols.[3] A *constraint* is an *equation* t_1=t_2, where t_1 and t_2 are terms, or an *inequation* $t_1 \odot t_2$, where t_1 and t_2 are arithmetic terms and $\odot \in \{$`<`,`>`,`<=`,`>=`$\}$. A *literal* is a defined predicate name together with a list of argument terms. Literals are sometimes considered as terms, i.e., the defined predicate names are also functor symbols. A *clause* has the form L `:-` L_1`,`$\ldots$`,`L_n where L is a literal and L_1`,`$\ldots$`,`L_n is a sequence of literals and constraints. For instance, the clauses of Examples 1 and 2 are CLP($\mathcal{R}$) programs in this sense.

The operational semantics of CLP($\mathcal{R}$) programs is similar to Prolog's operational semantics (SLD-resolution with leftmost selection rule) but with the difference that unification is replaced by adding a new equation between the literal and the clause head, and the computation proceeds only if all constraints (except the nonlinear) are satisfiable. Hence the set of constraints is divided into a set of *active constraints* and a set of *delayed constraints*. The constraint solver does only check the satisfiability of the active constraints. If a new constraint is generated during a computation step, it is added to the delayed constraints if it is nonlinear, otherwise to the active constraints. Moreover, delayed constraints are moved to the active constraint set if they become linear due to the addition of new active constraints. For instance, the delayed constraint `Z=X*Y` is activated if the new constraint `X=5` is added.

[1]In this simple example the groundness analysis can be improved by considering all constraints in an arbitrary order instead of a left-to-right order. However, this cannot be done in general if the constraints originate from the execution of several predicates.

[2]Similarly to CLP($\mathcal{R}$) [15] we assume that a CLP($\mathcal{R}$) program is well-typed.

[3]In contrast to CLP($\mathcal{R}$) we do not consider other arithmetic functions like `/`, `sin`, `cos`, `pow`, `abs`, `min` and `max`. These functions can be treated similarly to `*` in our abstract interpretation algorithm. We will discuss this subject in Section 5.

More details about the operational semantics and the delay mechanism can be found in [15, 11, 16]. The main goal of this paper is the characterization of a class of programs which have no delayed constraints at the end of a computation.

In order to keep our analysis simple, we transform CLP($\mathcal{R}$) programs into *flat* CLP($\mathcal{R}$) *programs* where each literal has the form $p(X_1, \ldots, X_n)$ (all X_i are distinct variables) and each constraint has one of the following forms (`X`, `Y`, $\mathtt{Y}_1,\ldots,\mathtt{Y}_n$, `Z` are variables, `c` is an atom or numeric constant and `f` is an uninterpreted functor symbol):

`X = Y`	`X = c`	$\mathtt{X = f(Y_1,\ldots,Y_n)}$				
`X = Y+Z`	`X = Y-Z`	`X = Y*Z`	`X < Y`	`X > Y`	`X <= Y`	`X >= Y`

It is obvious that every CLP($\mathcal{R}$) program can be transformed into a flat CLP($\mathcal{R}$) program by replacing terms by new variables and adding equations between the replaced terms and the corresponding new variables. This transformation does not change the principal answer behaviour. The only difference is that the transformed programs have more derivation steps (for the new equations) and additional equational constraints for the new variables. In the following we assume that all programs are flat CLP($\mathcal{R}$) programs.

3 Abstract interpretation of CLP($\mathcal{R}$) programs

In this section we present a method for the compile-time analysis of nonlinear constraints in CLP($\mathcal{R}$), i.e., a method for checking at compile time whether all nonlinear constraints become linear during the execution of the program. Obviously, a precise analysis requires a solution to the halting problem. Therefore we present an approximation to it based on an abstraction of the concrete behaviour of the program. If this approximation yields a positive answer, then it is ensured that all nonlinear constraints become linear at run time.

We assume familiarity with basic ideas of abstract interpretation techniques (see, for instance, the collection [1]). After the fundamental work of Cousot and Cousot [6] on systematic methods for program analysis several frameworks for the abstract interpretation of logic programs have been developed (see, for instance, [3, 18, 22]). These frameworks can also be used for the analysis of CLP($\mathcal{R}$) programs because of the similar operational semantics (SLD-resolution with left-to-right selection rule). The only differences to logic programming are:

- Substitutions are replaced by constraint sets. E.g., $\{\mathtt{X}\mapsto\mathtt{1},\mathtt{Y}\mapsto\mathtt{f(a)}\}$ can be represented by the constraint set $\{\mathtt{X=1},\mathtt{Y=f(a)}\}$.
- Unification of a literal L and a clause head H is replaced by adding the constraint L=H to the current constraint set. The existence of a unifier is then equivalent to the satisfiability of the extended constraint set.
- The composition of substitutions (e.g., combining the computed unifier with the previous substitution) is replaced by the conjunction of constraints.

Therefore we must define an appropriate abstraction of constraints (the abstract domain) and of constraint solving (the abstract operations). The correctness of the abstract interpretation algorithm can be proved by relating the abstractions to the concrete constraints. In the following we present the abstract domain and the abstract operations. The relation to concrete computations is presented in Section 4.

3.1 An abstract domain for analysing nonlinear constraints

The most important component of an abstract interpretation procedure is an abstract domain which approximates subsets of the concrete domain by finite representations. An element of the abstract domain describes common properties of a subset of the concrete domain. In our case the *concrete domain* is the set of all constraints where a constraint is a conjunction of equations and inequations. The important property of such a constraint is the existence of nonlinear elements in it. The precise form of these elements is not relevant for the analysis. Only the variable names in the nonlinear elements are needed in order to decide the potential linearity of these elements. Hence our abstract domain contains elements of the form $delay(\texttt{X}\ or\ \texttt{Y})$ representing a potential nonlinear constraint which will become linear if `X` or `Y` are constrained to unique values. Thus a correct abstraction of the constraint set

`X = Y*Z,   X = A*B,   T = A*C,   B = 3`

must contain the elements $delay(\texttt{Y}\ or\ \texttt{Z})$ and $delay(\texttt{A}\ or\ \texttt{C})$. It *may* also contain the element $delay(\texttt{A}\ or\ \texttt{B})$ if the information "`B` is unique" is not available. Note that the order of the variables in $delay(\texttt{X}\ or\ \texttt{Y})$ is not relevant, i.e., in the following we identify the elements $delay(\texttt{X}\ or\ \texttt{Y})$ and $delay(\texttt{Y}\ or\ \texttt{X})$.

To decide the potential linearity of these elements, we must know which variables are ground, i.e., which variables have unique values in all solutions of the constraint. A simple abstraction of groundness information of variables is a list of variables which are definitely ground [22] or an assignment of variables to the values *ground*, *free* or *any* [3]. However, this is not sufficient in our case since in constraint logic programming variables become ground not only by unification but also, and more important, by solving constraints when new constraints are added or delayed nonlinear constraints are awakened (like in CLP($\mathcal{R}$)). For instance, if the current constraints contain `X=Y*Z, T=3+Y`, then the addition of the new constraint `T=5` would cause `Y` to become ground and `Y*Z` to become linear. Hence our abstractions contain information about the dependencies between variables. To be more precise, our abstract domain contains elements of the form $V \Rightarrow \texttt{X}$ representing the fact that the variables in the set V uniquely determines the value of the variable `X`. As an extreme case, the abstraction element $\emptyset \Rightarrow \texttt{X}$ represents the fact that `X` has unique value, i.e., `X` is ground. For instance, an abstraction of the constraints `A=B+C, D=3+A` may contain the elements $\{\texttt{B,C}\} \Rightarrow \texttt{A}$, $\{\texttt{A,C}\} \Rightarrow \texttt{B}$, $\{\texttt{A,B}\} \Rightarrow \texttt{C}$, $\{\texttt{A}\} \Rightarrow \texttt{D}$, and $\{\texttt{D}\} \Rightarrow \texttt{A}$.

In our abstract interpretation algorithm we analyse the goal and each clause occurring in the program. The abstractions computed in this algorithm contain information about the variables in the goal or clause. Hence each abstraction A has a *domain* $dom(A)$ which is a set of variables occurring in some clause or goal. All variables occurring in A must belong to $dom(A)$.

Altogether, the *abstract domain* $\mathcal{A}$ contains the element $\perp$ (representing the empty subset of the concrete domain) and sets containing the following elements (such sets are called *abstractions* and denoted by A, A_1 etc):

Element:	*Meaning:*
$V \Rightarrow \texttt{X}$	the values of V determine the value of `X`
$delay(\texttt{X}\ or\ \texttt{Y})$	there is a delayed constraint which will be awakened if `X` or `Y` are ground, i.e., if `X` or `Y` have a unique value
$delay$	there is a delayed constraint which depends on arbitrary variables

Obviously, finiteness of $dom(A)$ imply finiteness of $\mathcal{A}$. The additional element *delay* is the "worst case" in the algorithm and will be used if the dependencies between nonlinear constraints and their variables are too complex for a finite representation. For convenience we simply write "X" instead of "$\emptyset \Rightarrow$X". Hence an abstraction element "X" means that variable X has a unique value.

For the sake of simplicity we will sometimes generate abstractions containing redundant information. The following *normalization rules* eliminate some redundancies in abstractions:

Normalization rules for abstractions:				
$A \cup \{Z,\ V \cup \{Z\} \Rightarrow X\}$	$\longrightarrow$	$A \cup \{Z,\ V \Rightarrow X\}$		(N1)
$A \cup \{X,\ delay(X\ or\ Y)\}$	$\longrightarrow$	$A \cup \{X\}$		(N2)
$A \cup \{V_1 \Rightarrow X,\ V_2 \Rightarrow X\}$	$\longrightarrow$	$A \cup \{V_1 \Rightarrow X\}$	if $V_1 \subseteq V_2$	(N3)

An abstraction A is called *normalized* if none of these normalization rules is applicable to A. Later we will see that the normalization rules are invariant w.r.t. the concrete constraints corresponding to abstractions. Therefore we can assume that we *compute only with normalized abstractions* in the abstract interpretation algorithm. In a concrete implementation one should add further normalization rules to delete other redundancies. However, the three rules above are sufficient for our intended results.

3.2 The abstract interpretation algorithm

The abstract interpretation algorithm is based on abstract operations corresponding to concrete operations during program execution. The most important concrete operations are the processing of a new constraint, the call of a clause for a predicate and the exit of a clause. In the following we describe the corresponding abstract operations.

First we describe the abstract processing of a new constraint. It is the most important operation in constraint logic programming and corresponds to unification in logic programming. At the abstract level it is a function $ai\text{-}con(\alpha, C)$ which takes an element of the abstract domain $\alpha \in \mathcal{A}$ and a single constraint C (equation or inequation) as input and produces another abstract domain element as the result. α is an abstraction of the possible given constraints and the result should be an abstraction of the given constraints together with the new constraint C. Since we are dealing with flat CLP($\mathcal{R}$) programs where all constraints have a restricted form (compare Section 2), it is sufficient to define *ai-con* by the following equations:

$$
\begin{array}{lll}
ai\text{-}con(\bot, C) & = \bot & \\
ai\text{-}con(A, \mathtt{X=Y}) & = A \cup \{\{\mathtt{X}\} \Rightarrow \mathtt{Y}, \{\mathtt{Y}\} \Rightarrow \mathtt{X}\} & \\
ai\text{-}con(A, \mathtt{X=c}) & = A \cup \{\mathtt{X}\} & \\
ai\text{-}con(A, \mathtt{X=f(Y_1,\ldots,Y}_n\mathtt{)}) & = A \cup \{\{\mathtt{Y}_1,\ldots,\mathtt{Y}_n\} \Rightarrow \mathtt{X}, \{\mathtt{X}\} \Rightarrow \mathtt{Y}_1, \ldots, \{\mathtt{X}\} \Rightarrow \mathtt{Y}_n\} & \\
ai\text{-}con(A, \mathtt{X=Y+Z}) & = A \cup \{\{\mathtt{Y,Z}\} \Rightarrow \mathtt{X},\ \{\mathtt{X,Z}\} \Rightarrow \mathtt{Y},\ \{\mathtt{X,Y}\} \Rightarrow \mathtt{Z}\} & \\
ai\text{-}con(A, \mathtt{X=Y-Z}) & = A \cup \{\{\mathtt{Y,Z}\} \Rightarrow \mathtt{X},\ \{\mathtt{X,Z}\} \Rightarrow \mathtt{Y},\ \{\mathtt{X,Y}\} \Rightarrow \mathtt{Z}\} & \\
ai\text{-}con(A, \mathtt{X=Y*Z}) & = A \cup \{\{\mathtt{Y,Z}\} \Rightarrow \mathtt{X},\ delay(\mathtt{Y}\ or\ \mathtt{Z})\} & \\
ai\text{-}con(A, \mathtt{X} \odot \mathtt{Y}) & = A & \text{if } \odot \in \{\mathtt{<}, \mathtt{>}, \mathtt{<=}, \mathtt{>=}\}
\end{array}
$$

The constraint X=Y implies a mutual dependency between both variables while the constraint X=f($\mathtt{Y}_1$,...,$\mathtt{Y}_n$) implies a dependency between X and the argument variables of the compound term. The variable X becomes ground by the constraint X=c while it become ground by the constraints X=Y+Z or X=Y-Z if two of the three variables are ground. The situation for X=Y*Z is a little bit different. Here X is ground if Y and Z are ground. But Y becomes ground only if X and Z are ground *and* $\mathtt{Z} \neq 0$. Since we have no access to the concrete values in our abstract domain, we cannot formulate this condition at the abstract level.[4] Similarly, we cannot express the fact that X becomes ground by the constraint X=Y*Z if Y or Z have a zero value. This is also the reason why inequations have no influence on the abstraction, i.e., *implicit equations* generated by inequations (e.g., the inequations X<=1,X>=1 generate the implicit equation X=1) are not detected at the abstract level.

Note that the function *ai-con* adds information to the current abstraction. The processing of this information (corresponding to constraint solving) is performed by the normalization rules. For instance, consider the goal

```
?- Z = X*Y,  U = V+X,  U = 5,  V = 3.
```

If we apply *ai-con* to the constraints from left to right starting with the empty abstraction, we obtain the abstraction

$\{\{\mathtt{X},\mathtt{Y}\}\Rightarrow\mathtt{Z},\ delay(\mathtt{X}\ or\ \mathtt{Y}),\ \{\mathtt{V},\mathtt{X}\}\Rightarrow\mathtt{U},\ \{\mathtt{U},\mathtt{X}\}\Rightarrow\mathtt{V},\ \{\mathtt{U},\mathtt{V}\}\Rightarrow\mathtt{X},\ \mathtt{U},\ \mathtt{V}\}$

which is not normalized. But this abstraction can be transformed by the normalization rules into

$\{\{\mathtt{Y}\}\Rightarrow\mathtt{Z},\ \mathtt{X},\ \mathtt{U},\ \mathtt{V}\}$

This normalized abstraction is a correct abstraction of the simplified answer constraint Z=2*Y,X=2,U=5,V=3. But note that $\{\mathtt{Z}\}\Rightarrow\mathtt{Y}$ is not contained in the last abstraction since the concrete value of X is not present in this abstraction.

We also need abstract operations for the abstract interpretation of defined predicates. The next operation restricts an abstraction A to a set of variables $W \subseteq dom(A)$. It will be used in a predicate call to omit the information about variables not passed from the predicate call to the applied clause:

$$call(\bot, W) = \bot$$
$$call(A, W) = \{V\Rightarrow\mathtt{X} \in A \mid \{\mathtt{X}\} \cup V \subseteq W\}$$

This operation also deletes all delay information in the given abstraction. This is justified since all omitted information is reconsidered after the predicate call (see below).

At the end of a clause a similar operation is necessary to forget the information about local clause variables. Hence we define:

$$\begin{aligned} exit(\bot, W) &= \bot \\ exit(A, W) &= \{V\Rightarrow\mathtt{X} \in A \mid \{\mathtt{X}\} \cup V \subseteq W\} \\ &\cup \{delay(\mathtt{X}\ or\ \mathtt{Y}) \in A \mid \mathtt{X},\mathtt{Y} \in W\} \\ &\cup \{delay \mid delay \in A \text{ or } delay(\mathtt{X}\ or\ \mathtt{Y}) \in A \text{ with } \{\mathtt{X},\mathtt{Y}\} \not\subseteq W\} \end{aligned}$$

This restriction operation for clause exits transforms an abstraction element $delay(\mathtt{X}\ or\ \mathtt{Y})$ into the element *delay* if one of the involved variables is not contained in W, i.e., it is noted that there may be a delayed constraint which depends on local variables at the end of the clause, but the possible dependencies are too complex for a finite abstract analysis. For a similar reason, the dependency $V\Rightarrow\mathtt{X}$ is simply omitted if $V \not\subseteq W$.

[4]This can be improved by including information about the sign of variables in our abstract domain. For instance, we could include (strict) inequalities between variables and the constant 0 as in the abstract domain *Ineq* of [17].

The *least upper bound* operation is used to combine the results of different clauses for a predicate call:

$$\begin{aligned}
\bot \sqcup A &= A \\
A \sqcup \bot &= A \\
A_1 \sqcup A_2 &= \{V_1 \cup V_2 \Rightarrow \mathtt{X} \mid V_1 \Rightarrow \mathtt{X} \in A_1,\ V_2 \Rightarrow \mathtt{X} \in A_2\} \\
&\quad \cup \{delay(\mathtt{X}\ or\ \mathtt{Y}) \mid delay(\mathtt{X}\ or\ \mathtt{Y}) \in A_1 \text{ or } delay(\mathtt{X}\ or\ \mathtt{Y}) \in A_2\} \\
&\quad \cup \{delay \mid delay \in A_1 \text{ or } delay \in A_2\}
\end{aligned}$$

Now we are able to present the algorithm for the abstract interpretation of a flat CLP($\mathcal{R}$) program. It is specified as a function $ai(\alpha, L)$ which takes an abstract domain element α and a literal or constraint L and yields a new abstract domain element as result. Clearly, $ai(\bot, L) = \bot$ and $ai(A, C) = ai\text{-}con(A, C)$ for all constraints C. The interesting case is the abstract interpretation of a defined predicate call $ai(A, p(X_1, \ldots, X_n))$ which is computed by the following steps ($var(\xi)$ denotes the set of all variables occurring in ξ):

1. Let $p(Z_1, \ldots, Z_n)$ `:-` $L_1, \ldots, L_k$ be a clause for predicate p (if necessary, rename the clause variables such that they are disjoint from $X_1, \ldots, X_n$)
   Compute: $A_{call} = call(A, \{X_1, \ldots, X_n\})$
   $A_0 = \langle \text{replace } X_i \text{ by } Z_i \text{ in } A_{call} \rangle$ $(dom(A_0) = \{Z_1, \ldots, Z_n\} \cup \bigcup_{i=1}^{k} var(L_i))$
   $A_1 = ai(A_0, L_1),\ A_2 = ai(A_1, L_2), \ldots, A_k = ai(A_{k-1}, L_k)$
   $A_{out} = exit(A_k, \{Z_1, \ldots, Z_n\})$
   $A_{exit} = \langle \text{replace all } Z_i \text{ by } X_i \text{ in } A_{out} \rangle$ (i.e., $dom(A_{exit}) = dom(A)$)
2. Let $A^1_{exit}, \ldots, A^m_{exit}$ be the exit substitutions of all clauses for p as computed in step 1. Then define $A_{success} = A^1_{exit} \sqcup \ldots \sqcup A^m_{exit}$
3. $ai(A, p(X_1, \ldots, X_n)) = A_{success} \cup (A - A_{call})$ if $A_{success} \neq \bot$, else $\bot$

Step 1 interprets a clause in the following way. Firstly, the *call abstraction* is computed, i.e., the information contained in the abstraction for the predicate call is restricted to the argument variables (A_{call}). The domain is changed to the clause variables by mapping argument variables to the corresponding variables of the applied clause (A_0). Then each literal in the clause body is interpreted. The resulting abstraction (A_k) is restricted to the variables in the clause head, i.e., we forget the information about the local variables in the clause. Potential delayed constraints which are not awakened at the clause end are passed to the abstraction A_{out} by the *exit* operation. In the last step the domain is changed to the original variables by renaming the variables of the clause head into the variables of the predicate call (A_{exit}). If all clauses defining the called predicate p are interpreted in this way, all possible interpretations are combined by the least upper bound of all abstractions ($A_{success}$). The combination of this abstraction with the information which was forgotten by the restriction at the beginning of the predicate call yields the abstraction after the predicate call (step 3).

Unfortunately, this abstract interpretation algorithm does not terminate in case of recursive programs. Since this problem is solved in all frameworks for abstract interpretation, we do not develop a new solution to this problem but we use one of the well-known techniques. Following Bruynooghe's framework [3] we construct a rational abstract AND-OR-tree representing the computation of the abstract interpretation algorithm. During the construction of the tree we check before the interpretation of a predicate call P

whether there is an ancestor node P' with a call to the same predicate and the same call abstraction (up to renaming of variables). If this is the case we take the success abstraction of P' (or $\perp$ if it is not available) as the success abstraction of P instead of interpreting P. If the further abstract interpretation computes a success abstraction A' for P' which differs from the success abstraction used for P, we start a recomputation beginning at P with A' as new success abstraction. This iteration terminates because all operations used in the abstract interpretation are monotone (w.r.t. the order on $\mathcal{A}$ defined in Section 4) and the abstract domain is finite. A detailed description of this method can be found in [3, 9].

3.3 Examples

The following flat CLP($\mathcal{R}$) program computes the product of all elements of a list of arithmetic expressions:

```
prod(A,B) :- A = [],     B = 1.
prod(A,B) :- A = [E|R],  B = E*P,  prod(R,P).
```

If we query this program with a list of numbers, as in

```
?- prod([2,3,4],Pr).
```

then the answer constraint is `Pr=24`. Our abstract interpretation algorithm computes the following abstractions for the initial goal `prod(L,Pr)` and the initial abstraction {`L`} (specifying the groundness of the first argument):

ai({`L`}, `prod(L,Pr)`) :
  Interpret the first clause:
    ai({`A`}, `A=[]`) = {`A`}
    ai({`A`}, `B=1`) = {`A`, `B`}
  Interpret the second clause:
    ai({`A`}, `A=[E|R]`) = {`A`, `E`, `R`}
    ai({`A`, `E`, `R`}, `B=E*P`) = {`A`, `E`, `R`, {`P`}$\Rightarrow$`B`}
    ai({`A`, `E`, `R`, {`P`}$\Rightarrow$`B`}, `prod(R,P)`):
    Recursive call:
    Take $\perp$ as result since success abstraction of ancestor call not available:
ai({`L`}, `prod(L,Pr)`) = {`L`, `Pr`}$\sqcup \perp$= {`L`, `Pr`}
    Recursive call `prod(R,P)` again:
    Take the new success abstraction {`R`, `P`} of ancestor call:
    ai({`A`, `E`, `R`, {`P`}$\Rightarrow$`B`}, `prod(R,P)`) = {`A`, `E`, `R`, {`P`}$\Rightarrow$`B`, `P`} $\rightarrow$ {`A`, `E`, `R`, `B`, `P`}
ai({`L`}, `prod(L,Pr)`) = {`L`, `Pr`} $\sqcup$ {`L`, `Pr`} = {`L`, `Pr`}

Hence the computed success abstraction is {`L`, `Pr`}. This means that after a successful computation of the goal `prod(L,Pr)` the variable `Pr` is bound to a ground term and there are no delayed constraints.

In a similar way one can compute the success abstraction of the goal `prod(L,Pr)` w.r.t. the initial abstraction {`Pr`}. The result is {`Pr`, *delay*} indicating that there may be a delayed constraint at the end of the concrete computation. In fact, the CLP($\mathcal{R}$) computation of the goal `?-prod([A,B,C],24)` produces the "maybe" nonlinear answer constraint `24=A*B*C`.

Similarly, our abstract interpretation algorithm computes the expected answers (w.r.t. to the delay information) to all queries shown in Example 1. More detailed examples can be found in [9].

4 Correctness of the analysis

In this section we will discuss the correctness of our abstract interpretation algorithm. As mentioned in Section 3 we use Bruynooghe's framework [3] for abstract interpretation of logic programs with the modifications listed in the beginning of Section 3. Therefore we have to relate the abstract domain to the concrete domain of constraints by defining a concretisation function, and then we have to prove the correctness of the abstract operations w.r.t. the corresponding operations on the concrete domain. Due to lack of space we omit the proofs of all theorems, but the interested reader will find them in [9].

The *concretisation function* $\gamma : \mathcal{A} \to 2^{\mathcal{C}}$ maps an abstraction into a subset of the concrete domain. In our case the *concrete domain* $\mathcal{C}$ is the set of all collections of constraints of the form

```
X = Y      X = c      X = f(Y1,...,Yn)
X = Y+Z    X = Y-Z    X = Y*Z     X < Y     X > Y     X <= Y     X >= Y
```

where `X`, `Y`, $\mathtt{Y}_1,\ldots,\mathtt{Y}_n$, `Z` are variables, `c` is an atom or numeric constant and `f` is an uninterpreted functor symbol. These are the constraints accumulated during the execution of a flat CLP($\mathcal{R}$) program and therefore sometimes called *flat constraints*. In practice a collection of such constraints is transformed into a simplified non-flat form in order to get a more efficient satisfiability check and readable answer constraints, but this is not relevant for our purpose. The meaning of a collection $C \in \mathcal{C}$ of constraints is the conjunction of all its elements, i.e., it specifies a set of solutions (mappings from variables into elements of the underlying constraint structure) satisfying each single constraint (cf. [13]):

$$Sol(C) \ := \{\sigma \mid \sigma \text{ is a valuation where } \sigma(c) \text{ is true for all } c \in C\}$$

The notion of "groundness" in logic programming corresponds to "uniqueness" of solutions in constraint logic programming. We say that variable X *is unique in the constraints* C if $\sigma_1(X) = \sigma_2(X)$ for all $\sigma_1, \sigma_2 \in Sol(C)$. Moreover, we say that a variable set V *determines* X *in* C if $\sigma_1(X) = \sigma_2(X)$ for all $\sigma_1, \sigma_2 \in Sol(C)$ with $\sigma_1 =_V \sigma_2$.[5] In this case we write $V \overset{C}{\Rightarrow} X$. Hence $\emptyset \overset{C}{\Rightarrow} X$ is equivalent to X unique in C. We call the arithmetic term `X*Y` *nonlinear in the constraints* C if both `X` and `Y` are not unique in C, i.e., a constraint containing this term would be delayed in CLP($\mathcal{R}$).

Now we are able to present the precise definition of the concretisation function $\gamma : \mathcal{A} \to 2^{\mathcal{C}}$ which relates an abstraction to a set of constraints:

$$\gamma(\perp) = \emptyset$$

$$\gamma(A) = \{C \in \mathcal{C} \mid \begin{array}{l} 1.\ V \overset{C}{\Rightarrow} \mathtt{X} \text{ for all } V \Rightarrow \mathtt{X} \in A \\ 2.\ \mathtt{X{=}Y{*}Z} \in C \text{ with } \mathtt{Y}, \mathtt{Z} \in dom(A) \text{ and } \mathtt{Y{*}Z} \text{ nonlinear in } C \\ \quad \Longrightarrow delay \in A \ \text{ or } \ delay(\mathtt{Y}\ or\ \mathtt{Z}) \in A \ \} \end{array}$$

The first condition expresses that for all abstraction elements $V \Rightarrow \mathtt{X} \in A$ the variables in V determine the value of `X` in all constraints corresponding to A. The second condition ensures that all nonlinear parts of constraints are contained in A. If this condition holds, we say that the nonlinear term `Y*Z` *is covered by* A. But note that only nonlinear terms having variables in the domain of A must be covered by A. This is due to the fact that A contains

[5] $\sigma_1 =_V \sigma_2$ is equivalent to the condition $\sigma_1(Z) = \sigma_2(Z)$ for all $Z \in V$.

abstract information about the variables of one clause but during the concrete computation the accumulated constraints may contain nonlinear parts from arbitrary clauses. Since we are interested in the analysis of *all* nonlinear constraints, we will show in Theorem 3 that the nonlinear constraints with variables outside $dom(A)$ are also covered by the abstraction A.

Since our abstract interpretation algorithm always simplifies the computed abstractions by the normalization rules of Section 3.1, it is important that these rules are invariant w.r.t. the concrete interpretation of abstractions. In fact, it can be shown that $\gamma(A) = \gamma(A')$ if $A \to A'$ [9].

All operations on the abstract domain must be monotone to ensure the the termination of the abstract interpretation algorithm. Hence we define the following order relation on normalized abstractions:

$$
\begin{array}{lll}
\text{(a)} & \bot \sqsubseteq \alpha \text{ for all } \alpha \in \mathcal{A} & \\
\text{(b)} & A \sqsubseteq A' \iff & 1.\ V' \Rightarrow X \in A' \implies \exists V \subseteq V' \text{ with } V \Rightarrow X \in A \\
 & & 2.\ delay(X \, or \, Y) \in A \;\Rightarrow\; delay(X \, or \, Y) \in A' \\
 & & 3.\ delay \in A \;\Rightarrow\; delay \in A'
\end{array}
$$

It is easy to prove that $\sqsubseteq$ is a reflexive, transitive and anti-symmetric relation on normalized abstractions. Moreover, the operation $\sqcup$ defined in Section 3.2 computes the least upper bound of two abstractions, and γ and all abstract operations defined in Section 3.2 are monotone functions.

Following the framework presented in [3], the correctness of the abstract interpretation algorithm can be proved by showing the correctness of each basic operation of the algorithm (like abstract constraint solving, clause entry and clause exit). *Correctness* means in this context that all concrete computations, i.e., the results of the concrete constraint solving, clause entry and clause exit (cf. Section 2) are subsumed by the abstractions computed by the corresponding abstract operations. The precise formulations and proofs of these properties can be found in [9].

There is one remaining problem with our abstract interpretation algorithm. The main motivation of this paper is the characterization of a class of CLP($\mathcal{R}$) programs where all nonlinear constraints become linear during the computation. If we analyse a CLP($\mathcal{R}$) program with our algorithm, the absence of *delay* elements in the success abstraction of the goal does not necessarily indicate that there are no delayed nonlinear constraints at the end of the computation. Due to the definition of our concretisation function γ, this indicates that there are no delayed nonlinear constraints containing goal variables. But it does not exclude the case that there are delayed constraints with variables local to some clauses. The next theorem shows that this case cannot occur since *all* delayed constraints are covered by our algorithm. We need the notion of "equivalence" of variables w.r.t. a constraint to formulate this theorem. We call two variables X, Y *equivalent w.r.t. constraint* C, denoted $X \sim_C Y$, if C constrains X and Y to the same values, i.e., $\sigma(X) = \sigma(Y)$ for all $\sigma \in Sol(C)$.

Theorem 3 (Completeness of delay abstractions) *Let L be a flat literal or constraint with abstraction A and $A' = ai(A, L)$. Let $C \in \gamma(A)$ and $C' \in \gamma(A')$ with $C' = C \cup C_L$ where C_L are the new constraints added to C during the execution of L. Then $delay \in A'$ or for all Z=Z_1*$Z_2 \in C_L$ with Z_1*Z_2 nonlinear in C' there exists $delay(Z_1' \, or \, Z_2') \in A'$ with $Z_1 \sim_{C'} Z_1'$ and $Z_2 \sim_{C'} Z_2'$.*

Due to this theorem our abstract interpretation algorithm characterizes a class of CLP($\mathcal{R}$) programs (those containing no new *delay* elements in the success abstraction of the goal) for which all nonlinear constraints become linear at run time.

5 Extension to other delayed constraints

In Section 2 we have defined the subclass of CLP($\mathcal{R}$) programs which we can analyse by our abstract interpretation algorithm. However, CLP($\mathcal{R}$) programs may also contain the arithmetic functions `/`, `sin`, `cos`, `pow`, `abs`, `min` and `max` which are also delayed until particular conditions are satisfied. For instance, the constraint `Z=sin(X)` is delayed until `X` is ground while the constraint `Z=abs(X)` is delayed until `X` is ground, `Z=0` or `Z` is ground and negative [11]. Since the exact value of a ground variable is not available in our abstract domain, we can only approximate this behaviour. In order to analyse these new constraints we have to extend our algorithm as follows:

1. Define a new element in the abstract domain appropriate to the abstract description of the delayed constraint.
2. Extend the abstract constraint solver *ai-con* to the new constraint.
3. Extend the normalization rules for abstractions to describe the wakeup conditions of the delayed constraint.

We demonstrate the necessary extensions by the new constraint `Z=min(X,Y)`. This constraint delays until `X` *and* `Y` are ground. Therefore we introduce the element $delay(\mathtt{X}\, and\, \mathtt{Y})$ in our abstract domain and extend the definition of *ai-con* to:

$$ai\text{-}con(A, \mathtt{Z{=}min(X,Y)}) \;=\; A \;\cup\; \{\; \{\mathtt{X,Y}\}\Rightarrow\mathtt{Z}, \quad delay(\mathtt{X}\; and\; \mathtt{Y}) \;\}$$

The wakeup condition for such constraints is described by the following normalization rule:

$$A \;\cup\; \{X,\; Y,\; delay(X\; and\; Y)\} \quad\longrightarrow\quad A \;\cup\; \{X,\; Y\}$$

All other types of delayed constraints can be handled in a similar way. Although we have not explicitly mentioned the necessary changes to *exit*, it is obvious how to adapt the definition of *exit* to the new kinds of constraints.

6 Applications

We have presented an algorithm to approximate the potential run-time occurrences of nonlinear constraints in a CLP($\mathcal{R}$) program. In this section we will outline possible applications of this algorithm.

6.1 Better user support

In CLP($\mathcal{R}$) the programmer can formulate arbitrary arithmetic constraints. However, during the computation process only linear arithmetic constraints are actively used to restrict the search space and control the computation. The programmer is responsible for writing the programs in such a way that all nonlinear constraints become linear during the computation. If this is not the case, the program may stop with a set of complex nonlinear constraints for which the satisfiability is difficult to decide. Unfortunately, it is not easy to see whether constraints become linear because this depends on the dataflow and the constraint solving in the program. Our algorithm is able to support the user in this difficult question since the algorithm can be applied in the following ways:

1. We start the algorithm with a particular goal and an initial abstraction. If the success abstraction computed for this goal contains no delay elements, then all computed answer constraints are linear, i.e., the CLP($\mathcal{R}$) constraint solver can decide the satisfiability of the final answer. Conditionally successful answers [11] cannot occur in this case.
2. If the user is interested not only in the final answer constraints but also in constraints produced during the computation process, we start the algorithm with a goal and an abstraction and consider at the end of the abstract interpretation the call and success abstractions of all literals in the program. Since these abstractions are valid approximations of all constraints which occur at run time, we can infer properties of intermediate constraints. For instance, if none of these abstractions contains a delay element, the programmer can be sure that the CLP($\mathcal{R}$) constraint solver decides the satisfiability of all constraints during the entire execution and thus useless derivations with unsolvable nonlinear constraints are not explored. On the other hand, delay elements in some abstraction indicate the program points where nonlinear constraints may occur at run time. This can be a useful information for the programmer.[6]

6.2 More efficient implementations

The knowledge about the potential presence of nonlinear constraints can be used to optimize the implementation of logic programs with arithmetic constraints. In this case it is necessary to consider the call and success abstractions of all literals rather than the success abstraction of the main goal (similarly to item 2 in Section 6.1 above). There are at least two potential optimizations:

1. If none of the abstractions contains a delay element, nonlinear constraints cannot occur at run time. Therefore general instructions for creating nonlinear constraints can be specialized to simpler instructions for creating linear constraints [14] and the program can be compiled without the delay mechanism for nonlinear constraints [16].
2. In the RISC-CLP(Real) system [12] nonlinear arithmetic constraints are not delayed but checked by a powerful constraint solver. But this constraint solver is very complex and sometimes too inefficient for solving simple linear constraints. We can optimize the RISC-CLP(Real) system by computing the program points where nonlinear constraints may occur and where all constraints are definitively linear. Then we can call a more efficient linear constraint solver for the latter program points without restricting the computational power of the RISC-CLP(Real) system.

6.3 Improving the termination behaviour

One of the principles of constraint logic programming is the satisfiability check during computation: a derivation proceeds only if all accumulated constraints are solvable [13]. This allows an early failure detection and avoids infinite derivation paths which may be present in pure logic programming. However,

[6] For such an application it may be necessary to change the definition of *call* so that delay elements are passed into the applied clause. Then the potential presence of nonlinear constraints can be immediately seen by considering the local abstraction without including the abstractions of ancestor nodes in the tree.

in CLP($\mathcal{R}$) this advantage is sometimes lost since nonlinear constraints are not checked for satisfiability. For instance, consider the following CLP($\mathcal{R}$) program for computing factorial numbers:

```
fac(0,1).
fac(N,N*F) :- N >= 1,  fac(N-1,F).
```

To compute a factorial we start with the goal `?-fac(8,F)` and obtain the answer constraint `F=40320`. If we want to know whether a given number is a factorial, we try to prove a goal like `?-fac(N,24)`. In this case CLP($\mathcal{R}$) computes the answer constraint `N=4` after some backtracking steps. Although nonlinear constraints are generated during this computation, they become linear if the first clause is used and binds the unknown first argument. But if we try to prove a (unsolvable) goal like `?-fac(N,10)`, CLP($\mathcal{R}$) runs into an infinite loop by applying the second clause again and again. The accumulated nonlinear constraints are not solvable but this is not detected by CLP($\mathcal{R}$) due to the delay mechanism. If we use a more powerful constraint solver which is able to treat nonlinear constraints (like in CAL [2] or RISC-CLP(Real) [12]), this infinite loop can be avoided.

We can use our abstract interpretation algorithm to find such sources of nontermination. For this purpose we compute the call abstraction of each literal in the program. If the abstraction of a recursive call contains a delay element, either we warn the user that there may be delayed nonlinear constraints before the recursive call (which can cause an infinite loop if these constraints are not solvable), or we use a powerful constraint solver for nonlinear constraints before the recursive call at run time in order to avoid the described source of nontermination. This seems to be a good compromise between the efficiency of the CLP($\mathcal{R}$) system and the power of the RISC-CLP(Real) system.

7 Conclusions and related work

We have presented a method for the analysis of nonlinear constraints occurring at run time in the execution of a CLP($\mathcal{R}$) program. Since an exact analysis is impossible at compile time, we have used an abstract interpretation algorithm to approximate the possible delayed nonlinear constraints and the variable dependencies occurring at run time. The application of this algorithm to various examples shows that our algorithm has enough precision for practical programs. The information produced by this algorithm can be used to support the programmer when using the delay mechanism of the CLP($\mathcal{R}$) system or to optimize the program when using a more powerful constraint solver like RISC-CLP(Real).

We have developed our analysis algorithm on the basis of a given framework for the abstract interpretation of logic programs [3] since the operational semantics of CLP($\mathcal{R}$) is very similar to logic programming. The only difference is the use of sets of constraints instead of substitutions. Therefore any other framework may also be applicable. Marriott and Søndergaard [19] have developed a particular framework for the abstract interpretation of constraint logic programming languages based on a denotational description of the semantics. They have also shown the application of their framework to the freeness and groundness analysis of CLP programs. However, they have not applied their method to a particular domain of constraints. Therefore they have not precisely described a solution to one of the main difficulties in a concrete application: the abstraction of the freeness or uniqueness of a variable w.r.t. a given concrete set of constraints. This is one of the main points addressed in this paper. We have derived uniqueness information w.r.t. arithmetic constraints over the real numbers by considering the variable dependencies caused by constraints. The normalization rules for our abstract domain corresponds to constraint solving in the concrete domain.

Most of the well-known abstract interpretation algorithms for the derivation of groundness information of variables or mode information for predicates in logic programs use a small number of abstract values like *ground*, *free* or *any* (see, for instance, [21, 22, 3] or [17] for the case of CLP($\mathcal{R}$)). Such a domain yields quite good results for many practical logic programs. But for constraint logic programming it must be refined since the possible reasons for the groundness of variables are much more complicated. For instance, the arithmetic constraint `X=Y+Z` implies the groundness of `Y` if `X` and `Z` are ground but *not* the groundness of `Y` and `Z` if `X` is ground. A typical programming methodology in constraint logic programming is "test and generate" [15, 23] where variables are instantiated by generators *after* the creation of a network of constraints between these variables. The following simple digital circuit program uses this technique (recall that we assume a left-to-right strategy for the evaluation of subgoals):

```
p(X,Y,Z) :- not(X,NX),and(NX,Y,NXY),not(Z,NZ),and(NXY,NZ,1),
            bit(X), bit(Y), bit(Z).          % generate
not(A,NA) :- NA = 1-A.
and(A,B,AB) :- AB = A*B.
bit(0).
bit(1).
```

The unique answer constraint to the goal `?-p(X,Y,Z)` is `X=0,Y=1,Z=0`, i.e., there are no delayed nonlinear constraints in the answer. However, a simple mode analysis as in [17] would infer that the predicate `and` is called with free variables in the first and second argument position and hence there may be a delayed nonlinear constraint at run time. In order to improve the accuracy of the analysis, we have used implications of the form $V \Rightarrow X$ to describe dependencies between different variables. For the last example our algorithm infer the dependencies {`X`}⇒`NX`, {`NX,Y`}⇒`NXY` and {`Z`}⇒`NZ` (among others). Since the variables `X`, `Y` and `Z` are bound to ground terms by the last `bit`-literals in the first clause, our algorithm infers (using the variable dependencies) that there are no delayed nonlinear constraints in the answer. This example shows that our algorithm has a better precision than other algorithms for groundness analysis which is due to the fact that grounding variables by constraint solving and awakening delayed constraints can be easily described on the abstract level with our abstract domain.

Marriott and Søndergaard have also proposed an abstract domain *Prop* for a more precise analysis of groundness information [20, 5]. Their domain contains propositional formulas over the program variables and the logical connectives $\vee$, $\wedge$ and $\leftrightarrow$. Such a domain is appropriate for pure logic programming since the groundness information after a unification like `X=f(Y,Z)` can be expressed by the propositional formula `X` $\leftrightarrow$ `Y` $\wedge$ `Z` meaning that the groundness of `X` is equivalent to the groundness of `Y` and `Z`. But in constraint logic languages the instantiation of variables may have different reasons as shown above by the constraint `X=Y+Z`. These different possibilities can be easily expressed in our abstract domain of variable dependencies.

Our abstract domain has some similarities to the abstract domain used for the analysis of residuating logic programs [10]. This is due to the fact that the analysis of variable dependencies is also essential for a precise analysis of residuating logic programs. However, the meaning of abstractions is quite different in both approaches. In case of residuating logic programs the concrete domain consists of substitutions and residuated equations and therefore substitutions must be interpreted w.r.t. the current set of residuated equations. In our case abstractions have a more direct meaning in the concrete domain and therefore the concretisation function and the correctness proofs are simpler. Further essential differences show up in the definition of abstract unification which is more sophisticated in the case of constraint logic programs.

Recently, García de la Banda and Hermenegildo [8] have independently developed a framework for the analysis of constraint logic programs by extending Bruynooghe's framework. Although they were mainly interested in the derivation of groundness information and did not include information about nonlinear constraints in their abstract domain, the abstract representation of variable dependencies is very similar to our approach. They also associate to each variable sets of variables which uniquely determine the value of that variable. However, they have given a direct definition of abstract constraint solving which results in more complicated definitions than our approach using normalization rules to simplify abstractions after abstract constraint solving.

Although our algorithm yields quite good results for practical programs, the precision of the uniqueness analysis can be improved in various ways. For instance, we do not consider the free variables in constraints and thus we do not detect the uniqueness of these variables in some cases. E.g., the constraint `3=5*X-2*X` restricts variable `X` to the unique value `1`. But our analysis algorithm does not infer that `X` is unique since the information that both subexpressions contain the same free variable is not present in the corresponding abstraction. Hence the analysis can be improved if the abstract domain is refined to store information about variables in expressions. Another possibility for improving the precision of the analysis is to derive information about possible values of variables. This would allow to detect that the constraints `X=3,6=X*Y` restricts `Y` to a unique value or that the constraints `X>2,Z=1,X<Z` are unsolvable.

Acknowledgements. This research was supported in part by the German Ministry for Research and Technology (BMFT) under grant ITS 9103 and by the ESPRIT Basic Research Working Group 6028 (Construction of Computational Logics). The responsibility for the contents of this publication lies with the author.

References

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[2] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. Constraint Logic Programming Language CAL. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 263–276, 1988.

[3] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming (10)*, pp. 91–124, 1991.

[4] A. Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, Vol. 33, No. 7, pp. 69–90, 1990.

[5] A. Cortesi, G. File, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. IEEE Symposium on Logic in Computer Science*, pp. 322–327, 1991.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252, 1977.

[7] S.K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 3, pp. 418–450, 1989.

[8] M.J. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. Technical Report, Universidad Politecnica de Madrid, 1992.

[9] M. Hanus. Analysis of Nonlinear Constraints in CLP($\mathcal{R}$). Technical Report MPI-I-92-251, Max-Planck-Institut für Informatik, Saarbrücken, 1992.

[10] M. Hanus. On the Completeness of Residuation. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming*, pp. 192–206. MIT Press, 1992.

[11] N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. *The CLP($\mathcal{R}$) Programmer's Manual, Version 1.1.* IBM Thomas J. Watson Research Center, Yorktown Heights, 1991.

[12] H. Hong. Non-linear Real Constraints in Constraint Logic Programming. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 201–212. Springer LNCS 632, 1992.

[13] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, pp. 111–119, Munich, 1987.

[14] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. An Abstract Machine for CLP($\mathcal{R}$). In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pp. 128–139. SIGPLAN Notices, Vol. 27, No. 7, 1992.

[15] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, Vol. 14, No. 3, pp. 339–395, 1992.

[16] J. Jaffar, S. Michaylov, and R.H.C. Yap. A Methodology for Managing Hard Constraints in CLP Systems. In *Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 306–316. SIGPLAN Notices, Vol. 26, No. 6, 1991.

[17] N. Jørgensen, K. Marriott, and S. Michaylov. Some Global Compile-Time Optimizations for CLP($\mathcal{R}$). In *Proc. 1991 International Logic Programming Symposium*, pp. 420–434. MIT Press, 1991.

[18] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and its Complexity Analysis. In *Proc. International Conference on Logic Programming*, pp. 64–78. MIT Press, 1991.

[19] K. Marriott and H. Søndergaard. Analysis of Constraint Logic Programs. In *Proc. of the 1990 North American Conference on Logic Programming*, pp. 531–547. MIT Press, 1990.

[20] K. Marriott, H. Søndergaard, and P. Dart. A Characterization of Non-Floundering Logic Programs. In *Proc. of the 1990 North American Conference on Logic Programming*, pp. 661–680. MIT Press, 1990.

[21] C.S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming (1)*, pp. 43–66, 1985.

[22] U. Nilsson. Towards a Framework for the Abstract Interpretation of Logic Programs. In *Proc. of the Workshop on Programming Language Implementation and Logic Programming*, pp. 68–82, Orléans, 1988. Springer LNCS 348.

[23] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.