In *New Generation Computing*, Vol. 13, No. 2, pp. 155–186, 1995

Compile-Time Analysis of Nonlinear Constraints in CLP($\mathcal{R}$)*

Michael Hanus
Informatik II, RWTH Aachen
D-52056 Aachen, Germany
hanus@informatik.rwth-aachen.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.

Keywords: Logic Programming, Constraints, Abstract Interpretation

1 Introduction

The constraint logic programming paradigm [15] 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($\mathcal{R}$) [17, 13] 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 [18] (this approach is also taken in Prolog III [5]). 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 1.1 Consider the following CLP($\mathcal{R}$) program to compute mortgage payments:

---

*This paper is an extended and revised version of [12]. The research described in this paper was made during the author's stay at the Max-Planck-Institut für Informatik in Saarbrücken, Germany. It 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.

```
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, 180, 0.01, 0, MP).
```

asks for the monthly payment to finance a mortgage, and the answer constraint is `MP=1200.17`. 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`. However, 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 CLP($\mathcal{R}$) system produces the "maybe" answer `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) [14] do not delay nonlinear constraints but apply special methods from computer algebra to check the satisfiability of all constraints.

Example 1.2 [14] 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).
```

If we ask the query `?-pyth(X,Y,Z)`, CLP($\mathcal{R}$) runs into an infinite loop since it does not detect that the linear and nonlinear constraints are not satisfiable, while RISC-CLP(Real) computes the answers `X=3,Y=4,Z=5`, `X=6,Y=8,Z=10`, and so on. □

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 [4, 8, 24]. 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 detailed 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 proved 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

[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($\mathcal{R}$) [17] we assume that a CLP($\mathcal{R}$) program is well-typed, i.e., variables, uninterpreted functors and predicates are used in such a way that arithmetic constraints do not contain "junk" like atoms at run time. However, this is not required in the formal description of CLP($\mathcal{R}$) programs due to the sake of simplicity. In order to check well-typedness at compile time we may extend the language with a type system for logic programming (see, for instance, [10] or the collection [26]).

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, ..., L_n where L is a literal and L_1, ..., L_n is a sequence of literals and constraints. For instance, the clauses of Examples 1.1 and 1.2 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. To give a precise definition of the operational semantics, a goal is written in the form C, D ?- G where C is a collection of satisfiable constraints (*active constraints*), D is a collection of *delayed constraints* and G is a sequence of literals and constraints. Initially, C and D are empty and G is the given goal.

A *computation step* is performed as follows. If the first (leftmost) element of G is a constraint, then it is moved to C if it is linear, otherwise to D. If the first element of G is a literal L, then it is deleted in G and the new equational constraint L=L_0 is generated where L_0 :- L_1, ..., L_n is (a new variant of) a program clause. Solving the constraint L=L_0 corresponds to unification in logic programming. Since L or L_0 may contain terms of the form X*Y, L=L_0 gives rise to a collection of constraints C'&D' where D' is the collection of nonlinear constraints. Hence C' is added to C and D' is added to D. A computation *fails* if the new set of linear constraints is unsatisfiable which is checked by the constraint solver.

It may be the case that a delayed constraint in D becomes linear due to the fact that the addition of new constraints in C implies the linearity of the delayed constraint because some of the variables in the initially nonlinear constraint get unique values. If this happens during a computation step, this delayed constraint is moved from D to C. More details about the operational semantics and the delay mechanism can be found in [13, 17, 18].

A derivation is called *successful* if both G and D are empty. If G is empty but D not, the derivation is called *conditionally successful* since it is not ensured that the constraints in D are satisfiable. The main goal of this paper is the characterization of a class of programs which have no conditionally successful derivations.

As an example for the operational semantics consider the initial goal

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

Then a successful derivation consists of the following elements:

[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.

C	D	?- G
$\emptyset$	$\emptyset$	?- Z=X*Y, X=A+B, C=3+A, B=5, C=6
$\emptyset$	{Z=X*Y}	?- X=A+B, C=3+A, B=5, C=6
{X=A+B}	{Z=X*Y}	?- C=3+A, B=5, C=6
{X=A+B, C=3+A}	{Z=X*Y}	?- B=5, C=6
{X=A+B, C=3+A, B=5}	{Z=X*Y}	?- C=6
{X=A+B, C=3+A, B=5, C=6, Z=X*Y}	$\emptyset$	?-

Note that the delayed constraint has been moved to the set of linear constraints in the last computation step because the constraints {X=A+B, C=3+A, B=5, C=6} imply the linearity of Z=X*Y. An equivalent but simplified form of the last constraint set is {X=8, A=3, B=5, C=6, Z=8*Y}.

In order to keep the abstract interpretation algorithm 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, $\mathtt{Y}_1$,...,$\mathtt{Y}_n$, Z are variables, c is an atom or numeric constant and f is an uninterpreted functor symbol):

```
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
```

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. For instance, consider the mortgage program in Example 1.1. This CLP($\mathcal{R}$) program is transformed into the following equivalent flat CLP($\mathcal{R}$) program:

```
mortgage(P,T,IR,B,MP) :-
        A = 0, T > A, C = 1, T <= C,
        D = T*IR, E = C+D, F = P*E,
        G = T*MP, B = F-G.
mortgage(P,T,IR,B,MP) :-
        A = 1, T > A,
        C = A+IR, D = P*C, E = D-MP, F = T-A,
        mortgage(E, F, IR, B, MP).
```

This transformation does not change the principal answer behavior. 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 behavior 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 [7] on systematic methods for program analysis several frameworks for the abstract interpretation of logic programs have been developed (see, for instance, [4, 20, 25]). 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 collections of constraints. E.g., the substitution $\{\mathtt{X}\mapsto\mathtt{1},\mathtt{Y}\mapsto\mathtt{f(a)}\}$ can be represented by the constraints $\{\mathtt{X=1,Y=f(a)}\}$.
- 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 the Analysis of 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 following properties of constraints are important for the analysis of nonlinear constraints:

1. Which variables are ground, i.e., which variables have unique values in all solutions of the constraint?
2. Which nonlinear elements are contained in the constraint?

The precise form of the nonlinear elements is not relevant for the analysis. Only the name of the variables in the nonlinear elements are important in order to decide the linearity of the elements. Therefore our abstract domain contains elements of the form

$$delay(\mathtt{X}\ or\ \mathtt{Y})$$

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 $delay(\texttt{Y}\ or\ \texttt{X})$.

A simple abstraction of groundness information of variables is a list of variables which are definitely ground [24] or an assignment of variables to the values *ground*, *free* or *any* [4]. 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 determine the value of the variable `X`. As an extreme case, the abstraction element $\emptyset \Rightarrow \texttt{X}$ represents the fact that `X` has a unique value, i.e., `X` is ground. For instance, an abstraction of the constraints `A=B+C, D=3+A` may contain the elements

$$\{\texttt{B,C}\} \Rightarrow \texttt{A},\ \{\texttt{A,C}\} \Rightarrow \texttt{B},\ \{\texttt{A,B}\} \Rightarrow \texttt{C},\ \{\texttt{A}\} \Rightarrow \texttt{D},\ \{\texttt{D}\} \Rightarrow \texttt{A}\ .$$

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, the finiteness of $dom(A)$ implies the 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 "$\emptyset \Rightarrow \texttt{X}$". Hence an element "`X`" in an abstraction 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)
$A \cup \{X,\ \mathit{delay}(X\ \mathit{or}\ Y)\}$	$\longrightarrow$	$A \cup \{X\}$		(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.

It is possible to add further normalization rules to delete some obvious redundancies, like

$$A \cup \{V \cup \{X\} \Rightarrow X\} \quad \longrightarrow \quad A$$
$$A \cup \{\mathit{delay}(X\ \mathit{or}\ Y),\ \mathit{delay}\} \quad \longrightarrow \quad A \cup \{\mathit{delay}\}$$

This should be done in a concrete implementation in order to keep the abstractions as small as possible. On the other hand, we could also add the rule

$$A \cup \{V_1 \Rightarrow X,\ V_2 \cup \{X\} \Rightarrow Y\} \quad \longrightarrow \quad A \cup \{V_1 \Rightarrow X,\ V_2 \cup \{X\} \Rightarrow Y,\ V_1 \cup V_2 \Rightarrow Y\}$$

(provided that $Y \notin V_1$) to add implicit dependencies to the abstraction which could improve the accuracy of the analysis in some examples. However, these rules are not necessary for the intended results in our examples. Therefore we omit such additional rules since it simplifies the correctness proofs in Section 4.

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 $\mathit{ai\text{-}con}(\alpha, C)$ 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:

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

The constraint `X=Y` implies a mutual dependency between both variables while the constraint $\mathtt{X=f(Y_1,\ldots,Y}_n\mathtt{)}$ 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 may 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

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

which is not normalized. But this abstraction is transformed by the normalization rules as follows:

$$
\begin{array}{ll}
\{\ \{\mathtt{X,Y}\}\Rightarrow\mathtt{Z},\ \textit{delay}(\mathtt{X}\ \textit{or}\ \mathtt{Y}),\ \{\mathtt{V,X}\}\Rightarrow\mathtt{U},\ \{\mathtt{U,X}\}\Rightarrow\mathtt{V},\ \{\mathtt{U,V}\}\Rightarrow\mathtt{X},\ \mathtt{U},\ \mathtt{V}\ \} & \\
\to \{\ \{\mathtt{X,Y}\}\Rightarrow\mathtt{Z},\ \textit{delay}(\mathtt{X}\ \textit{or}\ \mathtt{Y}),\ \{\mathtt{U,X}\}\Rightarrow\mathtt{V},\ \{\mathtt{U,V}\}\Rightarrow\mathtt{X},\ \mathtt{U},\ \mathtt{V}\ \} & \text{(by rule N3)} \\
\to \{\ \{\mathtt{X,Y}\}\Rightarrow\mathtt{Z},\ \textit{delay}(\mathtt{X}\ \textit{or}\ \mathtt{Y}),\ \{\mathtt{U,V}\}\Rightarrow\mathtt{X},\ \mathtt{U},\ \mathtt{V}\ \} & \text{(by rule N3)} \\
\to \{\ \{\mathtt{X,Y}\}\Rightarrow\mathtt{Z},\ \textit{delay}(\mathtt{X}\ \textit{or}\ \mathtt{Y}),\ \{\mathtt{V}\}\Rightarrow\mathtt{X},\ \mathtt{U},\ \mathtt{V}\ \} & \text{(by rule N1)} \\
\to \{\ \{\mathtt{X,Y}\}\Rightarrow\mathtt{Z},\ \textit{delay}(\mathtt{X}\ \textit{or}\ \mathtt{Y}),\ \mathtt{X},\ \mathtt{U},\ \mathtt{V}\ \} & \text{(by rule N1)} \\
\to \{\ \{\mathtt{X,Y}\}\Rightarrow\mathtt{Z},\ \mathtt{X},\ \mathtt{U},\ \mathtt{V}\ \} & \text{(by rule N2)} \\
\to \{\ \{\mathtt{Y}\}\Rightarrow\mathtt{Z},\ \mathtt{X},\ \mathtt{U},\ \mathtt{V}\ \} & \text{(by rule N1)}
\end{array}
$$

The last 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.

[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 [19].

We also need abstract operations for the analysis 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:

$$
\begin{array}{lcl}
\textit{call-restrict}(\perp, W) & = & \perp \\
\textit{call-restrict}(A, W) & = & \{V \Rightarrow \mathtt{X} \in A \mid \{\mathtt{X}\} \cup V \subseteq W\}
\end{array}
$$

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:

$$
\begin{array}{lcl}
\textit{exit-restrict}(\perp, W) & = & \perp \\
\textit{exit-restrict}(A, W) & = & \{V \Rightarrow \mathtt{X} \in A \mid \{\mathtt{X}\} \cup V \subseteq W\} \\
 & & \cup \{\textit{delay}(\mathtt{X}\ \textit{or}\ \mathtt{Y}) \in A \mid \mathtt{X}, \mathtt{Y} \in W\} \\
 & & \cup \{\textit{delay} \mid \textit{delay} \in A \text{ or } \textit{delay}(\mathtt{X}\ \textit{or}\ \mathtt{Y}) \in A \text{ with } \{\mathtt{X}, \mathtt{Y}\} \not\subseteq W\}
\end{array}
$$

This restriction operation for clause exits transforms an abstraction element $\textit{delay}(\mathtt{X}\ \textit{or}\ \mathtt{Y})$ 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 $V \not\subseteq W$.

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

$$
\begin{array}{ccccl}
\perp & \sqcup & A & = & A \\
A & \sqcup & \perp & = & A \\
A_1 & \sqcup & A_2 & = & \{V_1 \cup V_2 \Rightarrow \mathtt{X} \mid V_1 \Rightarrow \mathtt{X} \in A_1,\ V_2 \Rightarrow \mathtt{X} \in A_2\} \\
 & & & & \cup \{\textit{delay}(\mathtt{X}\ \textit{or}\ \mathtt{Y}) \mid \textit{delay}(\mathtt{X}\ \textit{or}\ \mathtt{Y}) \in A_1 \text{ or } \textit{delay}(\mathtt{X}\ \textit{or}\ \mathtt{Y}) \in A_2\} \\
 & & & & \cup \{\textit{delay} \mid \textit{delay} \in A_1 \text{ or } \textit{delay} \in A_2\}
\end{array}
$$

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, $ai(\perp, L) = \perp$ and $ai(A, C) = \textit{ai-con}(A, C)$ for all constraints C. The interesting case is the abstract interpretation of a call to a defined predicate, $ai(A, p(X_1, \ldots, X_n))$, which is computed by the following steps ($var(\xi)$ denotes the set of all variables occurring in the syntactic construction 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 $A_{call} = \textit{call-restrict}(A, \{X_1, \ldots, X_n\})$
   $A_0 = \langle$replace all X_i by Z_i in $A_{call}\rangle$ (i.e., $dom(A_0) = \{Z_1, \ldots, Z_n\} \cup \bigcup_{i=1}^{k} var(L_i)$)
   $A_1 = ai(A_0, L_1)$
   $A_2 = ai(A_1, L_2)$
   $\vdots$
   $A_k = ai(A_{k-1}, L_k)$

$A_{out} = exit\text{-}restrict(A_k, \{Z_1, \ldots, Z_n\})$
$A_{exit} = \langle$replace all Z_i by X_i 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 $A_{success} \neq \perp$, else $\perp$

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-restrict* 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 [4] we construct a rational abstract AND-OR-tree representing the computation of the abstract interpretation algorithm (see also Section 4.3). 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 is given in Section 4.3.

3.3 Examples

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

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

The corresponding flat CLP($\mathcal{R}$) program is:

```
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

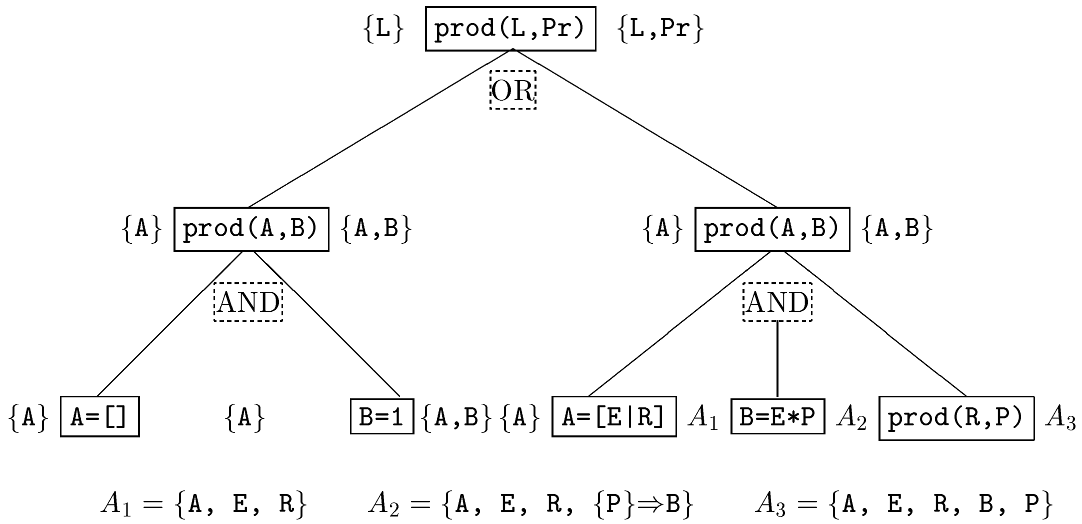

Figure 1: Final AND-OR-tree for the abstract interpretation of `prod(L,Pr)`

```
?- 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`}⇒`B`}
    ai({`A`, `E`, `R`, {`P`}⇒`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`}⇒`B`}, `prod(R,P)`) = {`A`, `E`, `R`, {`P`}⇒`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. The final AND-OR-tree for this abstract interpretation is shown in Figure 1 (the abstractions are written to the left and right of the corresponding literal).

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 ai({`Pr`},`prod(L,Pr)`) = {`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 1.1.

4 Correctness of the Abstract Interpretation Algorithm

In this section we will prove the correctness of the presented abstract interpretation algorithm. As mentioned in Section 3 we use Bruynooghe's framework [4] for abstract interpretation of logic programs with the modifications listed at the beginning of Section 3. Therefore we have to relate the abstract domain to the concrete domain of constraints by defining a concretisation function. If we can prove that the abstract operations defined in Section 3.2 are correct w.r.t. the corresponding operations on the concrete domain, the correctness of our algorithm is a direct consequence of Bruynooghe's work.

4.1 Relating Abstractions to Concrete Constraints

Our abstract interpretation algorithm is useless if we have no proposition about the relationship of the computed abstract properties of a flat CLP($\mathcal{R}$) program and the concrete constraints which can occur at run time. Therefore we have to define a *concretisation function* $\gamma{:}\mathcal{A} \to 2^{\mathcal{C}}$ which 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(Y`$_1$`,...,Y`$_n$`)`	
`X = Y+Z`	`X = Y-Z`	`X = Y*Z`	
`X < Y`	`X > Y`	`X <= Y`	`X >= Y`

where `X`, `Y`, `Y`$_1$,...,`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. [15]):

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

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

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

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 $\gamma{:}\mathcal{A} \rightarrow 2^{\mathcal{C}}$ which relates an abstraction to a set of constraints:

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

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. We also say that the constraints C *satisfy the variable condition* $V \Rightarrow \mathtt{X}$ if this condition holds. Hence $\mathtt{X} \in A$ implies that X is unique in all concrete 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 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 prove in Theorem 4.7 that the nonlinear constraints with variables outside $\mathit{dom}(A)$ 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, we have to show that these rules are invariant w.r.t. the concrete interpretation of abstractions. This is the purpose of the following lemma.

Lemma 4.1 *If A and A' are abstractions with $A \rightarrow A'$, then $\gamma(A) = \gamma(A')$.*

Proof: First we show $\gamma(A) \subseteq \gamma(A')$. Let $C \in \gamma(A)$. $C \in \gamma(A')$ can be proved by a case analysis on the applied normalization rule. We show only the nontrivial case N1, i.e., let $A = A_0 \cup \{Z,\ V \cup \{Z\} \Rightarrow X\}$ and $A' = A_0 \cup \{Z,\ V \Rightarrow X\}$. Since the only difference between A and A' is the transformation of "$V \cup \{Z\} \Rightarrow X$" into "$V \Rightarrow X$", we have to show $V \overset{C}{\Rightarrow} X$. Z is unique in C since $Z \in A$ and $C \in \gamma(A)$. Let $\sigma_1, \sigma_2 \in \mathit{Sol}(C)$ with $\sigma_1 =_V \sigma_2$. Then $\sigma_1 =_{V \cup \{Z\}} \sigma_2$ by uniqueness of Z in C. This implies $\sigma_1(X) = \sigma_2(X)$ by $V \cup \{Z\} \Rightarrow X \in A$. Hence $V \overset{C}{\Rightarrow} X$.

Next we show $\gamma(A) \supseteq \gamma(A')$. Let $C \in \gamma(A')$. As before $C \in \gamma(A)$ can be proved by a case analysis on the applied normalization rule but we show only the interesting rule N3. Let $A = A_0 \cup \{V_1 \Rightarrow X,\ V_2 \Rightarrow X\}$ and $A' = A_0 \cup \{V_1 \Rightarrow X\}$ with $V_1 \subseteq V_2$. We have to show that C satisfies the variable condition $V_2 \Rightarrow X$. Let $\sigma_1, \sigma_2 \in \mathit{Sol}(C)$ with $\sigma_1 =_{V_2} \sigma_2$. Then $\sigma_1 =_{V_1} \sigma_2$ since $V_1 \subseteq V_2$. Hence $\sigma_1(X) = \sigma_2(X)$ by $V_1 \Rightarrow X \in A'$ and $C \in \gamma(A')$. ∎

Due to this lemma it makes no difference to use an abstraction A or the normalization of A if we want to prove a proposition like $C \in \gamma(A)$. We will use this property in the correctness proofs for the abstract operations (cf. Section 4.2).

For the termination of the abstract interpretation algorithm it is important that all operations on the abstract domain are monotone. Therefore we define the following order relation on normalized abstractions:

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

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 which is a simple consequence of its definition.

Proposition 4.2 *$A_1 \sqcup A_2$ is a least upper bound of $A_1, A_2 \in \mathcal{A}$.*

It can also be easily shown that γ is a monotone function.

Proposition 4.3 *If $A \sqsubseteq A'$, then $\gamma(A) \subseteq \gamma(A')$.*

For the termination of the analysis algorithm it is essential that all abstract operations defined in Section 3.2 (call and exit restriction, abstract constraint solving etc.) are monotone. But this is also a direct consequence of the definition of $\sqsubseteq$ and thus we omit the simple proofs.

4.2 Correctness of Abstract Operations

Following the framework presented in [4], 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 concrete constraint solving, clause entry and clause exit (cf. Section 2) are subsumed by the abstractions computed by the corresponding abstract operations. In this section we will show the correctness of each of these operations.

The main result in this section is the correctness of *ai-con*, i.e., the abstract constraint solver *ai-con* covers all possible concrete constraints obtained by adding a new constraint to a given set of constraints. Since the proof requires a lengthy case distinction on the different kinds of constraints, the detailed proof is contained in the Appendix.

Theorem 4.4 (Correctness of abstract constraint solving) *Let A be an abstraction, c be a flat constraint (as defined in Section 4.1) with $var(c) \subseteq dom(A)$. Then $C \cup \{c\} \in \gamma(\mathit{ai\text{-}con}(A, c))$ for all $C \in \gamma(A)$.*

Next we want to prove that the abstract operations performed at the entry of a clause are correct w.r.t. the concrete operational semantics.

Theorem 4.5 (Correctness of clause entry) *Let $P = p(X_1, \ldots, X_n)$ be a predicate call with abstraction A and $C \in \gamma(A)$. Let $p(Z_1, \ldots, Z_n)$:- $L_1, \ldots, L_k$ be a (renamed) clause and A_0 be the abstraction computed by algorithm ai. Then $C \cup \{p(X_1, \ldots, X_n){=}p(Z_1, \ldots, Z_n)\} \in \gamma(A_0)$.*

Proof: Let A_0 be the abstraction after clause entry, i.e., A_0 is identical to A except that all delay abstractions are omitted and the variables are restricted to $\{X_1, \ldots, X_n\}$ and then renamed to $\{Z_1, \ldots, Z_n\}$. Hence $dom(A_0) = var(p(Z_1, \ldots, Z_n) \text{ :- } L_1, \ldots, L_k)$. Let $C' := C \cup \{p(X_1, \ldots, X_n)\texttt{=}p(Z_1, \ldots, Z_n)\}$. We have to show: $C' \in \gamma(A_0)$.

1. $V \Rightarrow \mathtt{Z} \in A_0$: Let ρ be the bijective renaming mapping $\rho = \{Z_1 \mapsto X_1, \ldots, Z_n \mapsto X_n\}$. By definition of A_0, $\rho(V) \Rightarrow \rho(\mathtt{Z}) \in A$. Let $\sigma_1, \sigma_2 \in Sol(C')$ with $\sigma_1 =_V \sigma_2$. Since σ_1 and σ_2 are solutions of $p(X_1, \ldots, X_n)\texttt{=}p(Z_1, \ldots, Z_n)$, $\sigma_1 =_{\rho(V)} \sigma_2$. Since $C \in \gamma(A)$ and $\rho(V) \Rightarrow \rho(\mathtt{Z}) \in A$, $\rho(V) \overset{C}{\Rightarrow} \rho(\mathtt{Z})$, and therefore $\sigma_1(\rho(\mathtt{Z})) = \sigma_2(\rho(\mathtt{Z}))$. This implies $\sigma_1(\mathtt{Z}) = \sigma_2(\mathtt{Z})$. Hence $V \overset{C'}{\Rightarrow} \mathtt{Z}$.
2. $\mathtt{X=Y*Z} \in C'$ with $\mathtt{Y}, \mathtt{Z} \in dom(A_0)$ and $\mathtt{Y*Z}$ nonlinear in C'. This case cannot occur since the only constraint in C' with variables from $dom(A_0)$ is $p(X_1, \ldots, X_n)\texttt{=}p(Z_1, \ldots, Z_n)$ (recall that the applied clause is a *new* variant and has no variables in common with the previous computation). ■

Next we prove the correctness of the abstract clause exit operations, i.e., we show that all constraints occurring at the end of a clause applied to a predicate call are covered by the abstract interpretation algorithm. Since the execution of a clause only adds new constraints to the constraints present at the beginning of the predicate call, it is sufficient to formulate the correctness criterion as in the following theorem.

Theorem 4.6 (Correctness of clause exit) *Let $P = p(X_1, \ldots, X_n)$ be a predicate call with abstraction A_{in} and $C_{in} \in \gamma(A_{in})$. Let $A = ai(A_{in}, P) = A_{success} \cup (A_{in} - A_{call})$ be the abstraction after the predicate call computed by the abstract interpretation algorithm ai. Let $L \text{ :- } L_1, \ldots, L_k$ be a (renamed) clause for P, and A_k be the abstraction computed for the clause end in ai. If $C_k \in \gamma(A_k)$ is an extension of C_{in} and the constraint $P\texttt{=}L$, i.e., $C_k = C_{in} \cup \{P\texttt{=}L\} \cup C$ for some constraints C, then $C_k \in \gamma(A)$.*

Proof: We show $C_k \in \gamma(A)$ if the conditions of the theorem are satisfied. Let $L = p(Z_1, \ldots, Z_n)$ and ρ be the bijective renaming mapping $\rho = \{X_1 \mapsto Z_1, \ldots, X_n \mapsto Z_n\}$.

1. $V \Rightarrow \mathtt{X} \in A$: We can distinguish two different cases:
   (a) $V \Rightarrow \mathtt{X} \in A_{in} - A_{call}$: Then $V \Rightarrow \mathtt{X} \in A_{in}$ and hence $V \overset{C_{in}}{\Rightarrow} \mathtt{X}$. This implies $V \overset{C_k}{\Rightarrow} \mathtt{X}$ because $C_k = C_{in} \cup \{P\texttt{=}L\} \cup C$.
   (b) $V \Rightarrow \mathtt{X} \in A_{success}$: By definition of $A_{success}$, there exists $V' \subseteq V$ with $\rho(V') \Rightarrow \rho(\mathtt{X}) \in A_k$ and $\rho(V') \cup \{\rho(\mathtt{X})\} \subseteq \{Z_1, \ldots, Z_n\}$. Hence $C_k \in \gamma(A_k)$ implies $\rho(V') \overset{C_k}{\Rightarrow} \rho(\mathtt{X})$. Since each solution of C_k is also a solution of $X_i\texttt{=}Z_i$ $(i = 1, \ldots, n)$, we obtain $V' \overset{C_k}{\Rightarrow} \mathtt{X}$ and therefore $V \overset{C_k}{\Rightarrow} \mathtt{X}$.
2. $\mathtt{X=Y*Z} \in C_k$ with $\mathtt{Y}, \mathtt{Z} \in dom(A)$ and $\mathtt{Y*Z}$ nonlinear in C_k. Since $C_k = C_{in} \cup \{P\texttt{=}L\} \cup C$ and C contains only new constraints with clause variables which are different from $dom(A)$, the constraint $\mathtt{X=Y*Z}$ must occur in C_{in}. Clearly, $\mathtt{Y*Z}$ is nonlinear in C_{in} since it is nonlinear in C_k. Therefore $C_{in} \in \gamma(A_{in})$ implies $delay \in A_{in}$ or $delay(\mathtt{Y}\ or\ \mathtt{Z}) \in A_{in}$. This delay abstraction is also contained in A because $A_{in} - A_{call} \subseteq A$ and A_{call} does not contain any delay abstractions (by definition of *call-restrict*). Hence the nonlinear term $\mathtt{Y*Z}$ is covered by A. ■

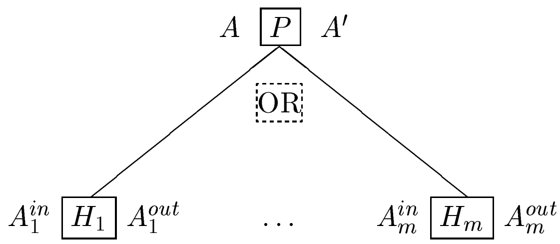

Figure 2: OR-node for clause entry

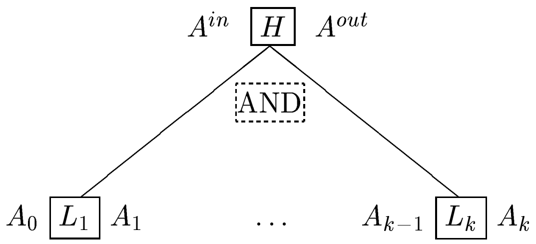

Figure 3: AND-node for a clause

4.3 Correctness of the Abstract Interpretation Algorithm

In the last section we have shown the local correctness of the three elementary operations of the abstract interpretation algorithm. Bruynooghe [4] has proved that these local correctness criteria are sufficient for the global correctness of the abstract interpretation algorithm. As already sketched in Section 3.2, the abstract interpretation algorithm generates an abstract AND-OR-tree which represents all concrete computations. To avoid infinite paths, this tree is a *rational* AND-OR-tree, i.e., if a predicate call is identical to (or a variant of) a predicate call in an ancestor node, then this call node is identified with the ancestor node and the abstract success information of the ancestor node is passed to this predicate call. Since the success abstraction of the predicate call may influence the success abstraction of the corresponding ancestor, the algorithm loops until these two abstractions are identical. The monotonicity property of all abstract operations together with the finite domain avoids an infinite looping in this graph. Now we present the abstract interpretation algorithm in more detail.

The abstract interpretation procedure generates the abstract AND-OR-graph as follows. In the first step, the root is created. It is marked with the initial goal (w.l.o.g. we assume that the initial goal contains only one literal) and the initial abstraction for this goal. Then this initial graph is extended by computing the success abstraction for this goal. The success abstraction A' of a single constraint c with abstraction A is computed by abstract constraint solving, i.e., $A' = \textit{ai-con}(A, c)$ (compare Section 3.2). We distinguish the following cases for the computation of the success abstraction A' of a node with a predicate call P and abstraction A:

1. There is no ancestor node with the same predicate call and the same call abstraction[6] (up to renaming of variables): First of all, we add an OR-node as shown in Figure 2 ($H_1, \ldots, H_m$ are

[6] Recall that the *call abstraction* of a predicate is the abstraction given before the predicate call restricted to the argument variables of the predicate call (compare operation *call-restrict* in Section 3.2).

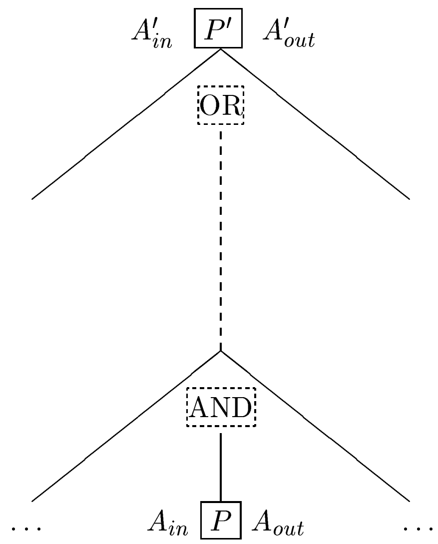

Figure 4: Recursive call: P is a renaming of P' and A_{in} restricted to call P is a renaming of A'_{in} restricted to call P'

the heads of all clauses for P). A_i^{in} is the abstraction computed by our abstract operations for the entry of clause H_i :- $\cdots$ (i.e., A_0 in algorithm *ai* in Section 3.2). Then for each new clause head H an AND-node is added as shown in Figure 3 where H :- $L_1, \ldots, L_k$ is the corresponding clause. After copying the abstraction of the head to the abstraction of the first body literal ($A_0 = A^{in}$) the success abstraction of each literal in the clause body is computed. Then the success abstraction A^{out} of the entire clause is calculated by restricting A_k to the head variables (i.e., A^{out} is identical to A_{out} in algorithm *ai* in Section 3.2). When all success abstractions of all clauses for the predicate call P are computed, they are renamed, combined by the least upper bound operation and then combined with the elements of abstraction A which were deleted in the call abstraction (compare algorithm *ai*).

2. There is an ancestor node P' with the same predicate call and the same call abstraction (up to renaming of variables) (Figure 4): Then the success abstraction of P' (A'_{out} without the elements already present in A'_{in}, i.e., $A_{success}$ in algorithm *ai* in Section 3.2) is taken as the success abstraction of P (or $\perp$ if it is not available). The combination of this success abstraction with the remaining elements of A_{in} yields A_{out} (step 3 of algorithm *ai*) and we proceed with the abstract interpretation procedure (i.e., we connect P to P'). If we reach the node P' at some point during the further computation and we compute a success abstraction for P' which differs from the old success abstraction taken for P, we recompute the success abstractions beginning at P where we take the new success abstraction of P' as new success abstraction for P. The monotonicity property of the abstract operations and the finite domain ensures that this iteration terminates.

Bruynooghe [4] has shown that this algorithm computes a superset of all concrete proof trees if the abstract operations for built-ins (here: constraints), clause entry and clause exit satisfies certain correctness conditions. We have mentioned at the beginning of Section 3 that this framework can also be applied to constraint logic programming if the notions of substitution and unification are replaced by constraints and constraint solving. Therefore Theorems 4.4, 4.5 and 4.6 imply exactly the necessary correctness conditions of Bruynooghe's framework applied to constraint logic programming. Hence we can infer the correctness of our abstract interpretation algorithm.

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.

Theorem 4.7 (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. If* `X=Y*Z` $\in C_L$ *with* `Y*Z` *nonlinear in C', then A' contains a delay element.*

Proof: If `X=Y*Z` $\in C_L$, this constraint must be added by executing a clause containing this constraint in the body (or $L =$ `X=Y*Z`, which is the trivial case). Since all concrete proof trees are represented by the abstract rational AND-OR-tree computed by the abstract interpretation algorithm (cf. [4]), this constraint must also be processed by our analysis algorithm which inserts the delay element *delay*(`Y` *or* `Z`). From the definition of *ai-con*, *exit-restrict*, $\sqcup$, and *ai* it is obvious that this delay element will never be deleted in the subsequent (success) abstractions. The only possibility to delete a delay element is an application of normalization rule N2, but this cannot happen if `Y*Z` is nonlinear in C' (and hence in all subsets of C') due to the correctness of the normalization rules (Lemma 4.1). Thus this delay element or a transformed version of it (by operation *exit-restrict* or renaming) is contained in A'. ■

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. A concrete example for the construction of an abstract AND-OR-tree has been shown in Section 3.3.

5 Extension to Other Delayed Constraints

In Section 2 we have defined the subclass of CLP($\mathcal{R}$) programs which can be analysed 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 [13]. Since the exact value of a ground variable is not available in our abstract domain, we can only approximate this behavior on the abstract level. 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.

In the following we demonstrate the necessary extensions by two examples.

`Z=sin(X)`: This constraint delays until `X` is ground. Therefore we introduce the element $delay(\texttt{X})$ in our abstract domain and extend the definition of *ai-con* to:

$$ai\text{-}con(A, \texttt{Z=sin(X)}) = A \cup \{\{\texttt{X}\}\Rightarrow\texttt{Z},\ delay(\texttt{X})\}$$

The wakeup condition for this kind of constraints is described by the following normalization rule for abstractions:

$$A \cup \{X,\ delay(X)\} \ \longrightarrow\ A \cup \{X\}$$

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

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

The wakeup condition for this kind of constraints is described by the following normalization rule:

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

All other types of delayed constraints can be handled in a similar way. Although we have not explicitly mentioned the necessary changes to *exit-restrict*, it is obvious how to adapt the definition of *exit-restrict* 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 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 (i.e., the entire abstract AND-OR-tree as shown in Section 3.3). 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, then the programmer can be sure that the CLP($\mathcal{R}$) constraint solver decides the satisfiability of all constraints during the entire execution and therefore 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.[7]

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 [16] and the program can be compiled without the delay mechanism for nonlinear constraints [18]. For instance, consider the following clause which may be part of an electronic circuit program.

   ```
   p(V,I,R) :- Rs = Rt+R, Rt = 2*R, V = Rs*I.
   ```

   Without any information about the run-time behavior of the program, the CLP($\mathcal{R}$) compiler translates the constraint `V = Rs*I` into an instruction which generates a possibly nonlinear constraint [16]. However, if it is known at compile time that predicate `p` will always be called such that the third argument `R` has a unique value, our algorithm infers that variable `Rs` has always a unique value just before the constraint `V = Rs*I` is processed. Hence the constraint `V = Rs*I` can be compiled into code for generating a linear constraint (see [16] for details) which has a more efficient behavior at run time.

[7] For such an application it may be necessary to change the definition of *call-restrict* 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.

2. The RISC-CLP(Real) system [14] allows the formulation of nonlinear arithmetic constraints which are not delayed but checked by a powerful constraint solver. However, this constraint solver is very complex and therefore sometimes too inefficient for solving simple linear constraints. Our algorithm can be used to optimize the RISC-CLP(Real) system since our algorithm determines the program points where nonlinear constraints may occur and where all constraints are definitively linear. Hence 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 Behavior

One of the principles of constraint logic programming is the satisfiability check during computation: a derivation proceeds only if all accumulated constraints are solvable [15]. This allows an early failure detection and avoids infinite derivation paths which may be present in pure logic programming. However, 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) [14]), 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, we may do the following:

1. 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.

2. 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.

If a solver for nonlinear constraints is integrated in the system, it should also be used at the end of a computation whenever the success abstraction of the initial goal contains delay elements.

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 [4] 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 [21] 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, [23, 24, 4] or [19] for the case of CLP($\mathcal{R}$)). Such a domain yields quite good results for many practical logic programs. However, 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" [17, 27] 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), % test
            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 [19] 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.

Our abstract domain has some similarities to the abstract domain used for the analysis of residuating logic programs [11]. 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.

García de la Banda and Hermenegildo [9] 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.

Recently, Baker and Søndergaard [3] have proposed to use the abstract domain *Prop* [22, 6] for a precise uniqueness analysis in constraint logic programs. Their domain consists of a particular subclass of propositional formulae over the program variables. The abstraction of arithmetic constraints is very similar to our approach, and their domain can additionally capture disjunctive information like "X or Y is definite." However, they have not considered the abstraction of delayed nonlinear constraints, and a rigorous correctness proof is not provided in their paper.

The main contribution of this paper is to provide an accurate analysis of nonlinear constraints together with a rigorous soundness proof of the analysis. The use of normalization rules on abstractions has simplified the correctness proof in comparison to a direct definition of an "abstract constraint solver". However, further work needs to be done in order to implement our algorithm in an efficient way. For instance, sophisticated data structures are required for the representation of our abstract domain in order to perform the normalization rules efficiently.

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. The author is grateful to Peter Barth, Veroniek Dumortier and Frank Zartmann for discussions and their careful reading of a previous version of this paper, and to two anonymous referees for their suggestions to improve its readability.

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] N. Baker and H. Søndergaard. Definiteness Analysis for CLP($\mathcal{R}$). Technical Report 92/25, Univ. of Melbourne, 1992.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of CLP Programs. In *Proc. International Logic Programming Symposium, Vancouver*, pp. 437–455. MIT Press, 1993.

[10] M. Hanus. Parametric Order-Sorted Types in Logic Programming. In *Proc. of the TAPSOFT '91*, pp. 181–200. Springer LNCS 494, 1991.

[11] 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.

[12] M. Hanus. Analysis of Nonlinear Constraints in CLP($\mathcal{R}$). In *Proc. Tenth International Conference on Logic Programming*, pp. 83–99. MIT Press, 1993.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] U. Nilsson. Systematic Semantic Approximations of Logic Programs. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 293–306. Springer LNCS 456, 1990.

[26] F. Pfenning, editor. *Types in Logic Programming.* MIT Press, 1992.

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

Appendix

Theorem 4.4 (Correctness of abstract constraint solving) Let A be an abstraction, c be a flat constraint (as defined in Section 4.1) with $var(c) \subseteq dom(A)$. Then $C \cup \{c\} \in \gamma(ai\text{-}con(A, c))$ for all $C \in \gamma(A)$.

Proof: First we prove the theorem for equational constraints. Let A be an abstraction, $\mathtt{X}{=}t$ be a flat constraint with $\{\mathtt{X}\} \cup var(t) \subseteq dom(A)$, and $C \in \gamma(A)$. We prove the theorem for each of the possible cases for t.

$\underline{t = \mathtt{Y}}$: Then

$$A' := ai\text{-}con(A, \mathtt{X{=}Y}) = A \cup \{\{\mathtt{X}\}\Rightarrow\mathtt{Y}, \{\mathtt{Y}\}\Rightarrow\mathtt{X}\}$$

We have to show: $C' := C \cup \{\mathtt{X{=}Y}\} \in \gamma(A')$.

1. $\{\mathtt{Y}\}\Rightarrow\mathtt{X} \in A'$: Let $\sigma_1, \sigma_2 \in Sol(C')$ with $\sigma_1(\mathtt{Y}) = \sigma_2(\mathtt{Y})$. Since σ_i is a solution of $\mathtt{X{=}Y}$, $\sigma_i(\mathtt{X}) = \sigma_i(\mathtt{Y})$ $(i = 1, 2)$. This implies $\sigma_1(\mathtt{X}) = \sigma_2(\mathtt{X})$.

2. $\{\mathtt{X}\}\Rightarrow\mathtt{Y} \in A'$: Symmetric to the previous case.

3. $V\Rightarrow\mathtt{Z} \in A$: Let $\sigma_1, \sigma_2 \in Sol(C')$ with $\sigma_1 =_V \sigma_2$. Since $\sigma_1, \sigma_2 \in Sol(C)$ and $C \in \gamma(A)$, $\sigma_1(\mathtt{Z}) = \sigma_2(\mathtt{Z})$. Hence $V \overset{C'}{\Rightarrow} \mathtt{Z}$.

   Therefore C' satisfies all variable conditions of A'.

4. Let $Z{=}Z_1{*}Z_2 \in C'$ with $Z_1, Z_2 \in dom(A)$ and $Z_1{*}Z_2$ nonlinear in C'. Clearly, $Z_1{*}Z_2$ is also nonlinear in C. Thus $C \in \gamma(A)$ implies that $delay \in A \subseteq A'$ or $delay(Z_1\ or\ Z_2) \in A \subseteq A'$. In any case the nonlinear term $Z_1{*}Z_2$ is also covered by A'.

$\underline{t = \mathtt{f(Y_1,\ldots,Y_n)}}$ with $\mathtt{f}$ an uninterpreted functor symbol: Then

$$A' := ai\text{-}con(A, \mathtt{X{=}f(Y_1,\ldots,Y_n)}) = A \cup \{\{\mathtt{Y_1,\ldots,Y_n}\}\Rightarrow\mathtt{X},\ \{\mathtt{X}\}\Rightarrow\mathtt{Y_1}, \ldots,\ \{\mathtt{X}\}\Rightarrow\mathtt{Y_n}\}$$

We have to show: $C' := C \cup \{\mathtt{X{=}f(Y_1,\ldots,Y_n)}\} \in \gamma(A')$.

1. $\{\mathtt{Y_1,\ldots,Y_n}\}\Rightarrow\mathtt{X} \in A'$: Let $\sigma_1, \sigma_2 \in Sol(C')$ with $\sigma_1 =_{\{\mathtt{Y_1,\ldots,Y_n}\}} \sigma_2$. Since σ_1 and σ_2 are solutions of $\mathtt{X{=}f(Y_1,\ldots,Y_n)}$,

$$\sigma_1(\mathtt{X}) = \sigma_1(\mathtt{f(Y_1,\ldots,Y_n)}) = \sigma_2(\mathtt{f(Y_1,\ldots,Y_n)}) = \sigma_2(\mathtt{X})$$

2. $\{\mathtt{X}\}\Rightarrow\mathtt{Y}_i \in A'$ for some $i \in \{1, \ldots, n\}$: Let $\sigma_1, \sigma_2 \in Sol(C')$ with $\sigma_1(\mathtt{X}) = \sigma_2(\mathtt{X})$. Since σ_1 and σ_2 are solutions of $\mathtt{X{=}f(Y_1,\ldots,Y_n)}$,

$$\sigma_1(\mathtt{f(Y_1,\ldots,Y_n)}) = \sigma_1(\mathtt{X}) = \sigma_2(\mathtt{X}) = \sigma_2(\mathtt{f(Y_1,\ldots,Y_n)})$$

   This equation implies $\sigma_1(\mathtt{Y}_i) = \sigma_2(\mathtt{Y}_i)$ because $\mathtt{f}$ is an uninterpreted functor symbol in the domain of CLP($\mathcal{R}$).

3. $V\Rightarrow\mathtt{Z} \in A$: This is identical to the case $t = \mathtt{Y}$.

4. Let $Z{=}Z_1{*}Z_2 \in C'$ with $Z_1, Z_2 \in dom(A)$ and $Z_1{*}Z_2$ nonlinear in C'. This case is also identical to the case $t = \mathtt{Y}$.

$\underline{t = \texttt{c}}$: This is a particular case of $t = \texttt{f(Y}_1\texttt{,...,Y}_n\texttt{)}$.
$\underline{t = \texttt{Y}_1\texttt{+Y}_2}$: Then

$$A' := \textit{ai-con}(A, \texttt{X=Y}_1\texttt{+Y}_2) = A \cup \{\{\texttt{Y}_1\texttt{,Y}_2\}\Rightarrow\texttt{X},\ \{\texttt{X,Y}_1\}\Rightarrow\texttt{Y}_2,\ \{\texttt{X,Y}_2\}\Rightarrow\texttt{Y}_1\}$$

We have to show: $C' := C \cup \{\texttt{X=Y}_1\texttt{+Y}_2\} \in \gamma(A')$.

1. $\{\texttt{Y}_1\texttt{,Y}_2\}\Rightarrow\texttt{X} \in A'$: Let $\sigma_1, \sigma_2 \in Sol(C')$ with $\sigma_1 =_{\{\texttt{Y}_1\texttt{,Y}_2\}} \sigma_2$. Since σ_1 and σ_2 are solutions of $\texttt{X=Y}_1\texttt{+Y}_2$,
$$\sigma_1(\texttt{X}) = \sigma_1(\texttt{Y}_1) + \sigma_1(\texttt{Y}_2) = \sigma_2(\texttt{Y}_1) + \sigma_2(\texttt{Y}_2) = \sigma_2(\texttt{X})$$
(note that $+$ is the addition function on real numbers while + is the syntactic denotation for an addition constraint). Hence $\{\texttt{Y}_1, \texttt{Y}_2\}\overset{C'}{\Rightarrow}\texttt{X}$.

2. $\{\texttt{X,Y}_1\}\Rightarrow\texttt{Y}_2 \in A'$: Let $\sigma_1, \sigma_2 \in Sol(C')$ with $\sigma_1 =_{\{\texttt{X,Y}_1\}} \sigma_2$. Since σ_1 and σ_2 are solutions of $\texttt{X=Y}_1\texttt{+Y}_2$
$$\sigma_1(\texttt{Y}_2) = \sigma_1(\texttt{X}) - \sigma_1(\texttt{Y}_1) = \sigma_2(\texttt{X}) - \sigma_2(\texttt{Y}_1) = \sigma_2(\texttt{Y}_2)$$
Hence $\{\texttt{X}, \texttt{Y}_1\}\overset{C'}{\Rightarrow}\texttt{Y}_2$.

3. $\{\texttt{X,Y}_2\}\Rightarrow\texttt{Y}_1 \in A'$: This is symmetric to the previous case.

4. $V\Rightarrow\texttt{Z} \in A$: This is identical to the case $t = \texttt{Y}$.

5. Let $Z\texttt{=}Z_1\texttt{*}Z_2 \in C'$ with $Z_1, Z_2 \in dom(A)$ and $Z_1\texttt{*}Z_2$ nonlinear in C'. This case is also identical to the case $t = \texttt{Y}$.

$\underline{t = \texttt{Y}_1\texttt{-Y}_2}$: Analogous to the case $t = \texttt{Y}_1\texttt{+Y}_2$.
$\underline{t = \texttt{Y}_1\texttt{*Y}_2}$: Then

$$A' := \textit{ai-con}(A, \texttt{X=Y}_1\texttt{*Y}_2) = A \cup \{\{\texttt{Y}_1\texttt{,Y}_2\}\Rightarrow\texttt{X},\ \textit{delay}(\texttt{Y}_1\ \textit{or}\ \texttt{Y}_2)\}$$

We have to show: $C' := C \cup \{\texttt{X=Y}_1\texttt{*Y}_2\} \in \gamma(A')$.

1. $\{\texttt{Y}_1\texttt{,Y}_2\}\Rightarrow\texttt{X} \in A'$: Analogous to the case $t = \texttt{Y}_1\texttt{+Y}_2$.

2. $V\Rightarrow\texttt{Z} \in A$: This is identical to the case $t = \texttt{Y}$.

3. Let $\texttt{X=Y}_1\texttt{*Y}_2 \in C'$ with $\texttt{Y}_1\texttt{*Y}_2$ nonlinear in C'. This nonlinear term is covered by A' because $\textit{delay}(\texttt{Y}_1\ \textit{or}\ \texttt{Y}_2) \in A'$.

4. Let $Z\texttt{=}Z_1\texttt{*}Z_2 \in C'$ with $Z\texttt{=}Z_1\texttt{*}Z_2 \neq \texttt{X=Y}_1\texttt{*Y}_2$, $Z_1, Z_2 \in dom(A)$ and $Z_1\texttt{*}Z_2$ nonlinear in C'. This is identical to the case $t = \texttt{Y}$.

It remains to prove the theorem for inequations. Let A be an abstraction, $\texttt{X}\odot\texttt{Y}$ be an inequation with $\odot \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\}$ and $\texttt{X}, \texttt{Y} \in dom(A)$, and $C \in \gamma(A)$. Since $\textit{ai-con}(A, \texttt{X}\odot\texttt{Y}) = A$, we have to show: $C' := C \cup \{\texttt{X}\odot\texttt{Y}\} \in \gamma(A)$.

1. $V\Rightarrow\texttt{Z} \in A$: Let $\sigma_1, \sigma_2 \in Sol(C')$ with $\sigma_1 =_V \sigma_2$. Since σ_1 and σ_2 are also solutions of C and $C \in \gamma(A)$, $\sigma_1(\texttt{Z}) = \sigma_2(\texttt{Z})$. Hence $V\overset{C'}{\Rightarrow}\texttt{Z}$.

2. Let $Z\texttt{=}Z_1\texttt{*}Z_2 \in C'$ with $Z_1, Z_2 \in dom(A)$ and $Z_1\texttt{*}Z_2$ nonlinear in C'. This case is also identical to the corresponding case for $\texttt{X=Y}$. ■