

MAX-PLANCK-INSTITUT FÜR INFORMATIK

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

Michael Hanus

MPI-I-92-251

November 1992



INFORMATIK

Im Stadtwald
W 6600 Saarbrücken
Germany

Author's Address

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

Acknowledgements

The research described in this paper 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.

Abstract

Solving nonlinear constraints over real numbers is a complex problem. Hence constraint logic programming languages like $\text{CLP}(\mathcal{R})$ 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- $\text{CLP}(\text{Real})$. Since such powerful constraint solvers are not very efficient, we propose a compromise between these two extremes. We characterize a class of $\text{CLP}(\mathcal{R})$ programs for which all delayed nonlinear constraints become linear at run time. Programs belonging to this class can be safely executed with the efficient $\text{CLP}(\mathcal{R})$ method while the remaining programs need a more powerful constraint solver.

Keywords

Logic Programming, Constraints, Abstract Interpretation

1 Introduction

The constraint logic programming paradigm [JL87] 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, $\text{CLP}(\mathcal{R})$ [JMSY92b, HJM⁺91] 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 $\text{CLP}(\mathcal{R})$ 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 [JMY91] (this approach is also taken in Prolog III [Col90]). 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 $\text{CLP}(\mathcal{R})$ 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, 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 $P=0.166783*B+83.3217*MP$. But if we want to compute the interest rate as in the query

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

$\text{CLP}(\mathcal{R})$ cannot compute a solved answer due to the restriction to linear constraints. The $\text{CLP}(\mathcal{R})$ system produces the “maybe” answer $600=(1000*IR+400)*(IR+1)$. \square

The $\text{CLP}(\mathcal{R})$ method of delaying nonlinear constraints and solving only linear constraints is efficient and successful for many problems. However, there are also problems where this method is not sufficient because $\text{CLP}(\mathcal{R})$ 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 [ASS⁺88] and RISC-CLP(Real) [Hon92] do

not delay nonlinear constraints but apply special methods from computer algebra to check the satisfiability of all constraints.

Example 1.2 [Hon92] 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)`, $\text{CLP}(\mathcal{R})$ runs into an infinite loop since it does not detect that the linear and nonlinear constraints are not satisfiable, while $\text{RISC-CLP}(\text{Real})$ computes the answers $X=3, Y=4, Z=5$, $X=6, Y=8, Z=10$, and so on. \square

Unfortunately, it is difficult to deal with nonlinear constraints, and constraint solvers for nonlinear constraints are not very efficient. Therefore it is undesirable to use such complex constraint solvers for problems which can be solved by the $\text{CLP}(\mathcal{R})$ method. Hence we propose a compromise between these two extremes. In the following we will characterize a class of $\text{CLP}(\mathcal{R})$ 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 $\text{CLP}(\mathcal{R})$ 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 $\text{CLP}(\mathcal{R})$ 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 [Bru91, Deb89, Nil88]. 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 addition of new constraints. For instance, consider the following sequence of constraints:

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

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 $\text{CLP}(\mathcal{R})$ programs for which our analysis is designed. The abstract domain and the abstract interpretation algorithm for the analysis of nonlinear constraints in $\text{CLP}(\mathcal{R})$ 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 $\text{CLP}(\mathcal{R})$ programs. Finally, we discuss possible applications of our method in Section 6.

¹In 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 Operational semantics of CLP(\mathcal{R}) programs

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

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 $L :- L_1, \dots, L_n$ where L is a literal and L_1, \dots, 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 $C, D \text{ ?- } G$ where C is a collection of satisfiable 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 $L_0 :- L_1, \dots, L_n$ 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 [JMSY92b, HJM⁺91, JMY91].

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

²Similarly to CLP(\mathcal{R}) [JMSY92b] 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, [Han91] or the collection [Pfe92]).

³In 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.

no conditionally successful derivations.

As an example for the operational semantics consider the following initial goal:

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

Then a successful derivation consists of the following elements:

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 $p(X_1, \dots, X_n)$ (all X_i are distinct variables) and each constraint has one of the following forms (X, Y, Y_1, \dots, Y_n, Z are variables, c is an atom or numeric constant and f is an uninterpreted functor symbol):

$$\begin{array}{lll}
X = Y & X = c & X = f(Y_1, \dots, Y_n) \\
X = Y+Z & X = Y-Z & X = Y*Z \\
X < Y & X > Y & X \leq Y \quad X \geq Y
\end{array}$$

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 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 [AH87]). After the fundamental work of Cousot and Cousot [CC77] on systematic methods for program analysis several frameworks for the abstract interpretation of logic programs have been developed (see, for instance, [Bru91, LCMVH91, Nil90]). 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 $\{X \mapsto 1, Y \mapsto f(a)\}$ can be represented by the constraints $\{X=1, Y=f(a)\}$.
- Unification of a goal 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

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

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

A simple abstraction of groundness information of variables is a list of variables which are definitely ground [Nil88] or an assignment of variables to the values *ground*, *free* or *any* [Bru91]. 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

$$V \Rightarrow X$$

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 $\emptyset \Rightarrow X$ 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

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

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 usually 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):

<i>Element:</i>	<i>Meaning:</i>
$V \Rightarrow X$	the values of V determine the value of X
$delay(X \text{ or } Y)$	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 $dom(A)$ imply 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 in the algorithm. For convenience we simply write “ X ” instead of “ $\emptyset \Rightarrow X$ ”. 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)
$A \cup \{X, \text{delay}(X \text{ or } Y)\}$	\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

$$\begin{aligned} A \cup \{V \cup \{X\} \Rightarrow X\} &\longrightarrow A \\ A \cup \{\text{delay}(X \text{ or } Y), \text{delay}\} &\longrightarrow A \cup \{\text{delay}\} \end{aligned}$$

This should be done in a concrete implementation in order to keep the abstractions as small as possible. However, these rules are not necessary for our intended results. Therefore we omit these 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 processing of a new constraint at the abstract level. 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 $ai\text{-}con$ by the following equations:

$$\begin{aligned} ai\text{-}con(\perp, C) &= \perp \\ ai\text{-}con(A, X=Y) &= A \cup \{\{X\} \Rightarrow Y, \{Y\} \Rightarrow X\} \\ ai\text{-}con(A, X=c) &= A \cup \{X\} \\ ai\text{-}con(A, X=f(Y_1, \dots, Y_n)) &= A \cup \{\{Y_1, \dots, Y_n\} \Rightarrow X, \{X\} \Rightarrow Y_1, \dots, \{X\} \Rightarrow Y_n\} \\ ai\text{-}con(A, X=Y+Z) &= A \cup \{\{Y, Z\} \Rightarrow X, \{X, Z\} \Rightarrow Y, \{X, Y\} \Rightarrow Z\} \\ ai\text{-}con(A, X=Y-Z) &= A \cup \{\{Y, Z\} \Rightarrow X, \{X, Z\} \Rightarrow Y, \{X, Y\} \Rightarrow Z\} \\ ai\text{-}con(A, X=Y*Z) &= A \cup \{\{Y, Z\} \Rightarrow X, \text{delay}(Y \text{ or } Z)\} \\ ai\text{-}con(A, X \odot Y) &= A \qquad \text{if } \odot \in \{<, >, <=, >=\} \end{aligned}$$

The constraint $X=Y$ implies a mutual dependency between both variables while the constraint $X=f(Y_1, \dots, Y_n)$ 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 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* $Z \neq 0$. 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

$$?- Z = X*Y, \quad U = V+X, \quad U = 5, \quad V = 3.$$

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

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

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

$$\begin{aligned} & \{ \{X, Y\} \Rightarrow Z, \text{ delay}(X \text{ or } Y), \{V, X\} \Rightarrow U, \{U, X\} \Rightarrow V, \{U, V\} \Rightarrow X, U, V \} \\ & \rightarrow \{ \{X, Y\} \Rightarrow Z, \text{ delay}(X \text{ or } Y), \{U, X\} \Rightarrow V, \{U, V\} \Rightarrow X, U, V \} && \text{(by rule N3)} \\ & \rightarrow \{ \{X, Y\} \Rightarrow Z, \text{ delay}(X \text{ or } Y), \{U, V\} \Rightarrow X, U, V \} && \text{(by rule N3)} \\ & \rightarrow \{ \{X, Y\} \Rightarrow Z, \text{ delay}(X \text{ or } Y), \{V\} \Rightarrow X, U, V \} && \text{(by rule N1)} \\ & \rightarrow \{ \{X, Y\} \Rightarrow Z, \text{ delay}(X \text{ or } Y), X, U, V \} && \text{(by rule N1)} \\ & \rightarrow \{ \{X, Y\} \Rightarrow Z, X, U, V \} && \text{(by rule N2)} \\ & \rightarrow \{ \{Y\} \Rightarrow Z, X, U, V \} && \text{(by rule N1)} \end{aligned}$$

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 $\{Z\} \Rightarrow Y$ 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 literals (defined predicates). The next operation restricts an abstraction A to a set of variables $W \subseteq \text{dom}(A)$. It will be used in a predicate call to omit the information about variables not passed from the predicate call to the applied clause:

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

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:

$$\text{exit-restrict}(\perp, W) = \perp$$

⁴This 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 [JMM91].

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

The restriction operation for clause exits transforms an abstraction element $\textit{delay}(\mathbf{X} \textit{ or } \mathbf{Y})$ into the element \textit{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.

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

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

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, $ai(\perp, L) = \perp$ and $ai(A, C) = ai\textit{-con}(A, C)$ for all constraints C . The interesting case is the abstract interpretation of a literal (defined predicate call) $ai(A, p(X_1, \dots, X_n))$ which is computed by the following steps ($var(\xi)$ denotes the set of all variables occurring in the syntactic construction ξ):

1. Let $p(Z_1, \dots, Z_n) :- L_1, \dots, L_k$ be a clause for predicate p
 (if necessary, rename the clause variables such that they are disjoint from X_1, \dots, X_n)
 Compute $A_{call} = call\textit{-restrict}(A, \{X_1, \dots, X_n\})$
 $A_0 = \langle \textit{replace all } X_i \textit{ by } Z_i \textit{ in } A_{call} \rangle$ (i.e., $dom(A_0) = \{Z_1, \dots, Z_n\} \cup \bigcup_{i=1}^k var(L_i)$)
 $A_1 = ai(A_0, L_1)$
 $A_2 = ai(A_1, L_2)$
 \vdots
 $A_k = ai(A_{k-1}, L_k)$
 $A_{out} = \textit{exit-restrict}(A_k, \{Z_1, \dots, Z_n\})$
 $A_{exit} = \langle \textit{replace all } Z_i \textit{ by } X_i \textit{ in } A_{out} \rangle$ (i.e., $dom(A_{exit}) = dom(A)$)
2. Let $A_{exit}^1, \dots, A_{exit}^m$ be the exit substitutions of all clauses for p as computed in step 1.
 Then define $A_{success} = A_{exit}^1 \sqcup \dots \sqcup A_{exit}^m$
3. $ai(A, p(X_1, \dots, X_n)) = A_{success} \cup (A - A_{call})$ 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 [Bru91] 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

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

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

$ai(\{L\}, \text{prod}(L, Pr))$:

Interpret the first clause:

$ai(\{A\}, A=[]) = \{A\}$

$ai(\{A\}, B=1) = \{A, B\}$

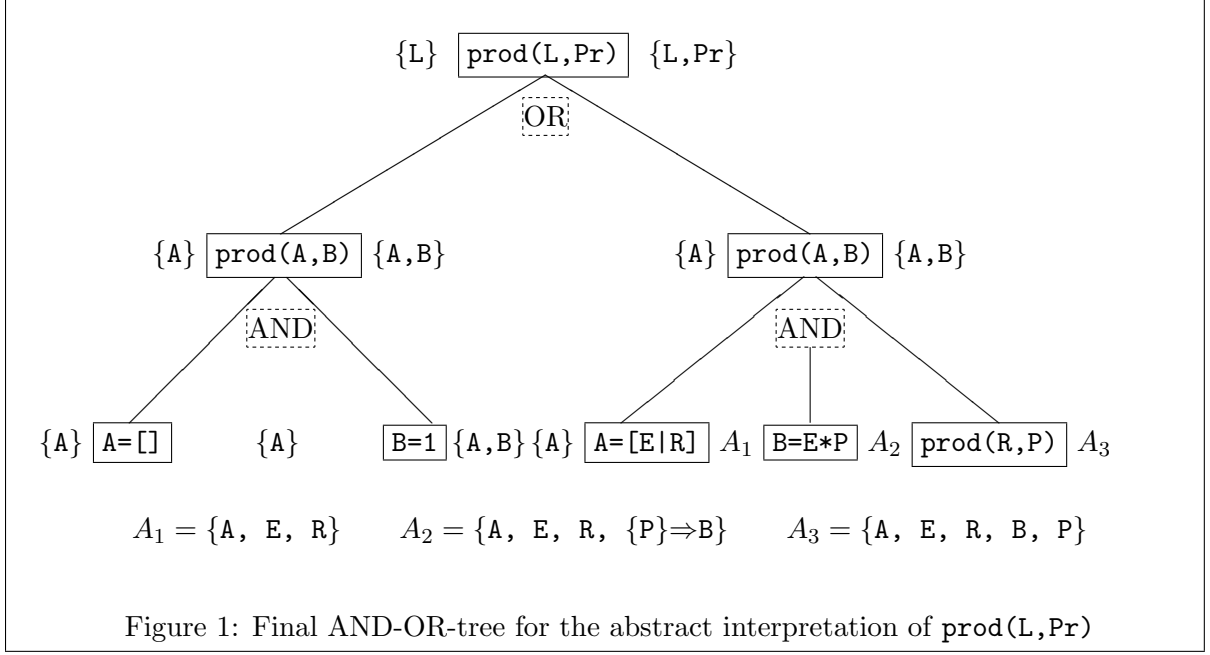
Interpret the second clause:

$ai(\{A\}, A=[E|R]) = \{A, E, R\}$

$ai(\{A, E, R\}, B=E*P) = \{A, E, R, \{P\} \Rightarrow B\}$

$ai(\{A, E, R, \{P\} \Rightarrow B\}, \text{prod}(R, P))$:

Recursive call: Take \perp as result since success abstraction of ancestor call not available:



$$ai(\{\text{L}\}, \text{prod}(\text{L}, \text{Pr})) = \{\text{L}, \text{Pr}\} \sqcup \perp = \{\text{L}, \text{Pr}\}$$

Recursive call $\text{prod}(\text{R}, \text{P})$ again: Take the new success abstraction $\{\text{R}, \text{P}\}$ of ancestor call:

$$ai(\{\text{A}, \text{E}, \text{R}, \{\text{P}\} \Rightarrow \text{B}\}, \text{prod}(\text{R}, \text{P})) = \{\text{A}, \text{E}, \text{R}, \{\text{P}\} \Rightarrow \text{B}, \text{P}\} \rightarrow \{\text{A}, \text{E}, \text{R}, \text{B}, \text{P}\}$$

$$ai(\{\text{L}\}, \text{prod}(\text{L}, \text{Pr})) = \{\text{L}, \text{Pr}\} \sqcup \{\text{L}, \text{Pr}\} = \{\text{L}, \text{Pr}\}$$

Hence the computed success abstraction is $\{\text{L}, \text{Pr}\}$. This means that after a successful computation of the goal $\text{prod}(\text{L}, \text{Pr})$ 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 $\text{prod}(\text{L}, \text{Pr})$ w.r.t. the initial abstraction $\{\text{Pr}\}$. The result is $ai(\{\text{Pr}\}, \text{prod}(\text{L}, \text{Pr})) = \{\text{Pr}, \text{delay}\}$ (compare Section 4.4) indicating that there may be a delayed constraint at the end of the concrete computation. In fact, the $\text{CLP}(\mathcal{R})$ computation of the goal

$$?- \text{prod}([\text{A}, \text{B}, \text{C}], 24).$$

produces the “maybe” nonlinear answer constraint $24 = \text{A} * \text{B} * \text{C}$.

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 [Bru91] 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. 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* $\gamma: \mathcal{A} \rightarrow 2^{\mathcal{C}}$ 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

$$\begin{array}{lll} \mathbf{X} = \mathbf{Y} & \mathbf{X} = \mathbf{c} & \mathbf{X} = \mathbf{f}(Y_1, \dots, Y_n) \\ \mathbf{X} = \mathbf{Y} + \mathbf{Z} & \mathbf{X} = \mathbf{Y} - \mathbf{Z} & \mathbf{X} = \mathbf{Y} * \mathbf{Z} \\ \mathbf{X} < \mathbf{Y} & \mathbf{X} > \mathbf{Y} & \mathbf{X} \leq \mathbf{Y} \qquad \mathbf{X} \geq \mathbf{Y} \end{array}$$

where $\mathbf{X}, \mathbf{Y}, Y_1, \dots, Y_n, \mathbf{Z}$ are variables, \mathbf{c} is an atom or numeric constant and \mathbf{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. [JL87]):

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

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 $V \stackrel{C}{\Rightarrow} X$. Hence $\emptyset \stackrel{C}{\Rightarrow} X$ is equivalent to X unique in C . We call the arithmetic term $\mathbf{X} * \mathbf{Y}$ *nonlinear in the constraints* C if both \mathbf{X} and \mathbf{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:

$$\begin{aligned} \gamma(\perp) &= \emptyset \\ \gamma(A) &= \{ C \in \mathcal{C} \mid \begin{array}{l} 1. V \stackrel{C}{\Rightarrow} \mathbf{X} \text{ for all } V \Rightarrow \mathbf{X} \in A \\ 2. \mathbf{X} = \mathbf{Y} * \mathbf{Z} \in C \text{ with } \mathbf{Y}, \mathbf{Z} \in dom(A) \text{ and } \mathbf{Y} * \mathbf{Z} \text{ nonlinear in } C \\ \implies delay \in A \text{ or } delay(\mathbf{Y} \text{ or } \mathbf{Z}) \in A \} \end{array} \end{aligned}$$

The first condition expresses that for all abstraction elements $V \Rightarrow \mathbf{X} \in A$ the variables in V determine the value of \mathbf{X} in all constraints corresponding to A . We also say that the constraints C *satisfy the variable condition* $V \Rightarrow \mathbf{X}$ if this condition holds. Hence $\mathbf{X} \in A$ implies that \mathbf{X} is unique in all concrete constraints corresponding to A .

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

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 4.8 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, 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 $C \in \gamma(A)$. We prove $C \in \gamma(A')$ by a case analysis on the applied normalization rule:

N1: 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 $V \stackrel{C}{\Rightarrow} X$. Z is unique in C since $Z \in A$ and $C \in \gamma(A)$. Let $\sigma_1, \sigma_2 \in Sol(C)$ 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 $V \stackrel{C}{\Rightarrow} X$.

N2: Let $A = A_0 \cup \{X, delay(X \text{ or } Y)\}$ and $A' = A_0 \cup \{X\}$. If the abstraction element $delay(X \text{ or } Y)$ covers the nonlinear term $X*Y$ in C , then X is not unique in C . But this is a contradiction to $X \in A$. Therefore C cannot contain the nonlinear term $X*Y$.

N3: Let $A = A_0 \cup \{V_1 \Rightarrow X, V_2 \Rightarrow X\}$ and $A' = A_0 \cup \{V_1 \Rightarrow X\}$ with $V_1 \subseteq V_2$. Obviously, $C \in \gamma(A')$ since the variable condition $V_2 \Rightarrow X$ is omitted in A' .

Next we show $\gamma(A) \supseteq \gamma(A')$. Let $C \in \gamma(A')$. As before we prove $C \in \gamma(A)$ by a case analysis on the applied normalization rule:

N1: Let $A = A_0 \cup \{Z, V \cup \{Z\} \Rightarrow X\}$ and $A' = A_0 \cup \{Z, V \Rightarrow X\}$. Obviously, $C \in \gamma(A)$ since $V \stackrel{C}{\Rightarrow} X$ implies $V \cup \{Z\} \stackrel{C}{\Rightarrow} X$.

N2: Let $A = A_0 \cup \{X, delay(X \text{ or } Y)\}$ and $A' = A_0 \cup \{X\}$. This case is trivial since A contains the additional abstraction element “ $delay(X \text{ or } Y)$ ”.

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 that variable condition $V_2 \Rightarrow X$. Let $\sigma_1, \sigma_2 \in Sol(C)$ 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:

- (a) $\perp \sqsubseteq \alpha$ for all $\alpha \in \mathcal{A}$
- (b) $A \sqsubseteq A' \iff$
 1. $V' \Rightarrow X \in A' \implies \exists V \subseteq V'$ with $V \Rightarrow X \in A$
 2. $\text{delay}(X \text{ or } Y) \in A \implies \text{delay}(X \text{ or } Y) \in A'$
 3. $\text{delay} \in A \implies \text{delay} \in A'$

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:

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

Proof: We assume $A_1 \neq \perp$ and $A_2 \neq \perp$, otherwise the lemma is obviously true.

First we show that $A_1 \sqcup A_2$ is an upper bound of A_1 (the case for A_2 is symmetric): Let $V \Rightarrow X \in A_1 \sqcup A_2$. By definition of \sqcup , there are $V_1 \Rightarrow X \in A_1$ and $V_2 \Rightarrow X \in A_2$ with $V = V_1 \cup V_2$. Hence $V_1 \Rightarrow X \in A_1$ and $V_1 \subseteq V$ (condition 1 of \sqsubseteq). If α is an abstraction element of the form $\text{delay}(X \text{ or } Y)$ or delay , then $\alpha \in A_1$ implies $\alpha \in A_1 \sqcup A_2$ by definition of \sqcup (conditions 2, 3 of \sqsubseteq). Therefore $A_1 \sqsubseteq A_1 \sqcup A_2$.

To show that $A_1 \sqcup A_2$ is a least upper bound, assume an abstraction A with $A_1 \sqsubseteq A$ and $A_2 \sqsubseteq A$. If $V \Rightarrow X \in A$, then there are $V_1 \subseteq V$ and $V_2 \subseteq V$ with $V_1 \Rightarrow X \in A_1$ and $V_2 \Rightarrow X \in A_2$ (by definition of \sqsubseteq). This implies $V_1 \cup V_2 \Rightarrow X \in A_1 \sqcup A_2$ and $V_1 \cup V_2 \subseteq V$. If δ is an abstraction element of the form $\text{delay}(X \text{ or } Y)$ or delay , then $\delta \in A_1 \sqcup A_2$ implies $\delta \in A_1$ or $\delta \in A_2$ and hence $\delta \in A$ by definition of \sqsubseteq . Therefore $A_1 \sqcup A_2 \sqsubseteq A$. ■

It is also easy to show that γ is a monotone function:

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

Proof: Let $A \sqsubseteq A'$ and $C \in \gamma(A)$. (the case $A = \perp$ is trivial). We have to show $C \in \gamma(A')$.

Let $V' \Rightarrow X \in A'$. Since $A \sqsubseteq A'$, there is a set $V \subseteq V'$ with $V \Rightarrow X \in A$. Since $C \in \gamma(A)$, $V \stackrel{C}{\Rightarrow} X$. This immediately implies $V' \stackrel{C}{\Rightarrow} X$. Hence C satisfies $V' \Rightarrow X$.

Let $\mathbf{X}=\mathbf{Y}*\mathbf{Z} \in C$ with $\mathbf{Y}, \mathbf{Z} \in \text{dom}(A)$ and $\mathbf{Y}*\mathbf{Z}$ nonlinear in C . Since $C \in \gamma(A)$, A contains a corresponding delay element covering this nonlinear term. But this delay element is also contained in A' by definition of \sqsubseteq . ■

It is also not difficult to show that all abstract operations defined in Section 3.2 (call and exit restriction, abstract constraint solving etc.) are monotone. As an example we show the monotonicity of the restriction operation for clause exits.

Lemma 4.4 The abstract operation *exit-restrict* is monotone.

Proof: Let $A_1 \sqsubseteq A_2$ and $A'_1 := \text{exit-restrict}(A_1, W)$, $A'_2 := \text{exit-restrict}(A_2, W)$. We have to show: $A'_1 \sqsubseteq A'_2$.

If $A_1 = \perp$, then $A'_1 = \perp$ and thus $A'_1 \sqsubseteq A'_2$. Hence we assume $A_1 \neq \perp$ which implies $A_2 \neq \perp$ and $A'_1 \neq \perp, A'_2 \neq \perp$.

1. $V \Rightarrow X \in A'_2$: By definition of *exit-restrict*, $V \Rightarrow X \in A_2$ and $\{X\} \cup V \subseteq W$. Since $A_1 \sqsubseteq A_2$, there is a set $V_1 \subseteq V$ with $V_1 \Rightarrow X \in A_1$. Hence $V_1 \Rightarrow X \in A'_1$ because $\{X\} \cup V_1 \subseteq \{X\} \cup V \subseteq W$.
2. $\text{delay}(X \text{ or } Y) \in A'_1$: By definition of *exit-restrict*, $\text{delay}(X \text{ or } Y) \in A_1$ and $X, Y \in W$. This implies $\text{delay}(X \text{ or } Y) \in A_2$ and thus $\text{delay}(X \text{ or } Y) \in A'_2$.
3. $\text{delay} \in A'_1$: By definition of *exit-restrict*, either $\text{delay} \in A_1$ which implies $\text{delay} \in A_2$ and $\text{delay} \in A'_2$, or $\text{delay}(X \text{ or } Y) \in A_1$ with $\{X, Y\} \not\subseteq W$. The latter case implies $\text{delay}(X \text{ or } Y) \in A_2$ and $\text{delay} \in A'_2$.

■

4.2 Correctness of abstract operations

Following the framework presented in [Bru91], 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. In this section we will prove the correctness of each of these operations.

First we prove that the abstract constraint solving *ai-con* covers all possible concrete constraints obtained by adding a new constraint to a given set of constraints.

Theorem 4.5 (Correctness of abstract constraint solving) *Let A be an abstraction, c be a flat constraint (as defined in Section 4.1) with $\text{var}(c) \subseteq \text{dom}(A)$. Then $C \cup \{c\} \in \gamma(\text{ai-con}(A, c))$ for all $C \in \gamma(A)$.*

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

$t = \mathbf{Y}$: Then

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

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

1. $\{\mathbf{Y}\} \Rightarrow \mathbf{X} \in A'$: Let $\sigma_1, \sigma_2 \in \text{Sol}(C')$ with $\sigma_1(\mathbf{Y}) = \sigma_2(\mathbf{Y})$. Since σ_i is a solution of $\mathbf{X}=\mathbf{Y}$, $\sigma_i(\mathbf{X}) = \sigma_i(\mathbf{Y})$ ($i = 1, 2$). This implies $\sigma_1(\mathbf{X}) = \sigma_2(\mathbf{X})$.
2. $\{\mathbf{X}\} \Rightarrow \mathbf{Y} \in A'$: Symmetric to the previous case.
3. $V \Rightarrow \mathbf{Z} \in A$: Let $\sigma_1, \sigma_2 \in \text{Sol}(C')$ with $\sigma_1 =_V \sigma_2$. Since $\sigma_1, \sigma_2 \in \text{Sol}(C)$ and $C \in \gamma(A)$, $\sigma_1(\mathbf{Z}) = \sigma_2(\mathbf{Z})$. Hence $V \stackrel{C'}{\Rightarrow} \mathbf{Z}$.

Therefore C' satisfies all variable conditions of A' .

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

$t = \mathbf{f}(Y_1, \dots, Y_n)$ with \mathbf{f} an uninterpreted functor symbol: Then

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

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

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

$$\sigma_1(\mathbf{X}) = \sigma_1(\mathbf{f}(Y_1, \dots, Y_n)) = \sigma_2(\mathbf{f}(Y_1, \dots, Y_n)) = \sigma_2(\mathbf{X})$$

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

$$\sigma_1(\mathbf{f}(Y_1, \dots, Y_n)) = \sigma_1(\mathbf{X}) = \sigma_2(\mathbf{X}) = \sigma_2(\mathbf{f}(Y_1, \dots, Y_n))$$

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

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

$t = \mathbf{c}$: This is a particular case of $t = \mathbf{f}(Y_1, \dots, Y_n)$.

$t = Y_1+Y_2$: Then

$$A' := ai-con(A, \mathbf{X}=Y_1+Y_2) = A \cup \{\{Y_1, Y_2\} \Rightarrow \mathbf{X}, \{\mathbf{X}, Y_1\} \Rightarrow Y_2, \{\mathbf{X}, Y_2\} \Rightarrow Y_1\}$$

We have to show: $C' := C \cup \{\mathbf{X}=Y_1+Y_2\} \in \gamma(A')$.

1. $\{Y_1, Y_2\} \Rightarrow \mathbf{X} \in A'$: Let $\sigma_1, \sigma_2 \in Sol(C')$ with $\sigma_1 =_{\{Y_1, Y_2\}} \sigma_2$. Since σ_1 and σ_2 are solutions of $\mathbf{X}=Y_1+Y_2$,

$$\sigma_1(\mathbf{X}) = \sigma_1(Y_1) + \sigma_1(Y_2) = \sigma_2(Y_1) + \sigma_2(Y_2) = \sigma_2(\mathbf{X})$$

(note that $+$ is the addition function on real numbers while $+$ is the syntactic denotation for an addition constraint). Hence $\{Y_1, Y_2\} \stackrel{C'}{\Rightarrow} \mathbf{X}$.

2. $\{\mathbf{X}, Y_1\} \Rightarrow Y_2 \in A'$: Let $\sigma_1, \sigma_2 \in Sol(C')$ with $\sigma_1 =_{\{\mathbf{X}, Y_1\}} \sigma_2$. Since σ_1 and σ_2 are solutions of $\mathbf{X}=Y_1+Y_2$

$$\sigma_1(Y_2) = \sigma_1(\mathbf{X}) - \sigma_1(Y_1) = \sigma_2(\mathbf{X}) - \sigma_2(Y_1) = \sigma_2(Y_2)$$

Hence $\{\mathbf{X}, Y_1\} \stackrel{C'}{\Rightarrow} Y_2$.

3. $\{\mathbf{X}, Y_2\} \Rightarrow Y_1 \in A'$: This is symmetric to the previous case.
4. $V \Rightarrow Z \in A$: This is identical to the case $t = Y$.
5. 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 $t = Y$.

$t = Y_1 - Y_2$: Analogous to the case $t = Y_1 + Y_2$.

$t = Y_1 * Y_2$: Then

$$A' := ai-con(A, X=Y_1*Y_2) = A \cup \{\{Y_1, Y_2\} \Rightarrow X, delay(Y_1 \text{ or } Y_2)\}$$

We have to show: $C' := C \cup \{X=Y_1*Y_2\} \in \gamma(A')$.

1. $\{Y_1, Y_2\} \Rightarrow X \in A'$: Analogous to the case $t = Y_1 + Y_2$.
2. $V \Rightarrow Z \in A$: This is identical to the case $t = Y$.
3. Let $X=Y_1*Y_2 \in C'$ with Y_1*Y_2 nonlinear in C' . This nonlinear term is covered by A' because $delay(Y_1 \text{ or } Y_2) \in A'$.
4. Let $Z=Z_1*Z_2 \in C'$ with $Z=Z_1*Z_2 \neq X=Y_1*Y_2$, $Z_1, Z_2 \in dom(A)$ and Z_1*Z_2 nonlinear in C' . This is identical to the case $t = Y$.

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

1. $V \Rightarrow Z \in A$: 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 $C \in \gamma(A)$, $\sigma_1(Z) = \sigma_2(Z)$. Hence $V \stackrel{C'}{\Rightarrow} Z$.
2. 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 corresponding case for $X=Y$.

■

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

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 $\{X_1, \dots, X_n\}$ and then renamed to $\{Z_1, \dots, Z_n\}$. Hence $dom(A_0) = var(p(Z_1, \dots, Z_n) :- L_1, \dots, L_k)$. Let $C' := C \cup \{p(X_1, \dots, X_n) = p(Z_1, \dots, Z_n)\}$. We have to show: $C' \in \gamma(A_0)$.

1. $V \Rightarrow Z \in A_0$: Let ρ be the bijective renaming mapping $\rho = \{Z_1 \mapsto X_1, \dots, Z_n \mapsto X_n\}$. By definition of A_0 , $\rho(V) \Rightarrow \rho(Z) \in A$. Let $\sigma_1, \sigma_2 \in Sol(C')$ with $\sigma_1 =_V \sigma_2$. Since σ_1 and σ_2 are solutions of $p(X_1, \dots, X_n) = p(Z_1, \dots, Z_n)$, $\sigma_1 =_{\rho(V)} \sigma_2$. Since $C \in \gamma(A)$ and $\rho(V) \Rightarrow \rho(Z) \in A$, $\rho(V) \stackrel{C}{\Rightarrow} \rho(Z)$, and therefore $\sigma_1(\rho(Z)) = \sigma_2(\rho(Z))$. This implies $\sigma_1(Z) = \sigma_2(Z)$. Hence $V \stackrel{C'}{\Rightarrow} Z$.
2. $X=Y*Z \in C'$ with $Y, Z \in dom(A_0)$ and $Y*Z$ nonlinear in C' . This case cannot occur since the only constraint in C' with variables from $dom(A_0)$ is $p(X_1, \dots, X_n) = p(Z_1, \dots, Z_n)$ (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 4.7 (Correctness of clause exit) *Let $P = p(X_1, \dots, X_n)$ 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 $L :- L_1, \dots, L_k$ 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 $P=L$, i.e., $C_k = C_{in} \cup \{P=L\} \cup C$ 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 $L = p(Z_1, \dots, Z_n)$ and ρ be the bijective renaming mapping $\rho = \{X_1 \mapsto Z_1, \dots, X_n \mapsto Z_n\}$.

1. $V \Rightarrow \mathbf{X} \in A$: We can distinguish two different cases:

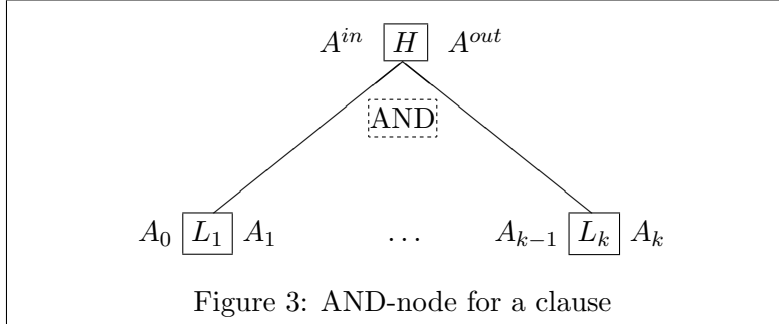
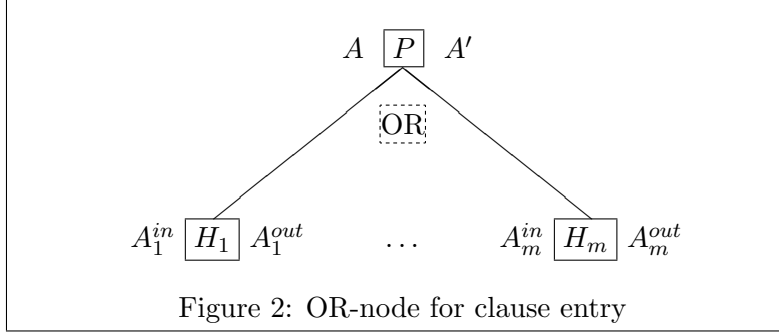
- (a) $V \Rightarrow \mathbf{X} \in A_{in} - A_{call}$: Then $V \Rightarrow \mathbf{X} \in A_{in}$ and hence $V \xrightarrow{C_{in}} \mathbf{X}$. This implies $V \xrightarrow{C_k} \mathbf{X}$ because $C_k = C_{in} \cup \{P=L\} \cup C$.
- (b) $V \Rightarrow \mathbf{X} \in A_{success}$: By definition of $A_{success}$, there exists $V' \subseteq V$ with $\rho(V') \Rightarrow \rho(\mathbf{X}) \in A_k$ and $\rho(V') \cup \{\rho(\mathbf{X})\} \subseteq \{Z_1, \dots, Z_n\}$. Hence $C_k \in \gamma(A_k)$ implies $\rho(V') \xrightarrow{C_k} \rho(\mathbf{X})$. Since each solution of C_k is also a solution of $X_i = Z_i$ ($i = 1, \dots, n$), we obtain $V' \xrightarrow{C_k} \mathbf{X}$ and therefore $V \xrightarrow{C_k} \mathbf{X}$.

2. $\mathbf{X} = \mathbf{Y} * \mathbf{Z} \in C_k$ with $\mathbf{Y}, \mathbf{Z} \in dom(A)$ and $\mathbf{Y} * \mathbf{Z}$ nonlinear in C_k . Since $C_k = C_{in} \cup \{P=L\} \cup C$ and C contains only new constraints with clause variables which are different from $dom(A)$, the constraint $\mathbf{X} = \mathbf{Y} * \mathbf{Z}$ must occur in C_{in} . Clearly, $\mathbf{Y} * \mathbf{Z}$ 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 $delay(\mathbf{Y} \text{ or } \mathbf{Z}) \in A_{in}$. 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 $\mathbf{Y} * \mathbf{Z}$ is covered by A .

■

4.3 Correctness of the abstract interpretation algorithm

In the last section we have proved the local correctness of the three elementary operations of the abstract interpretation algorithm. Bruynooghe [Bru91] has shown 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

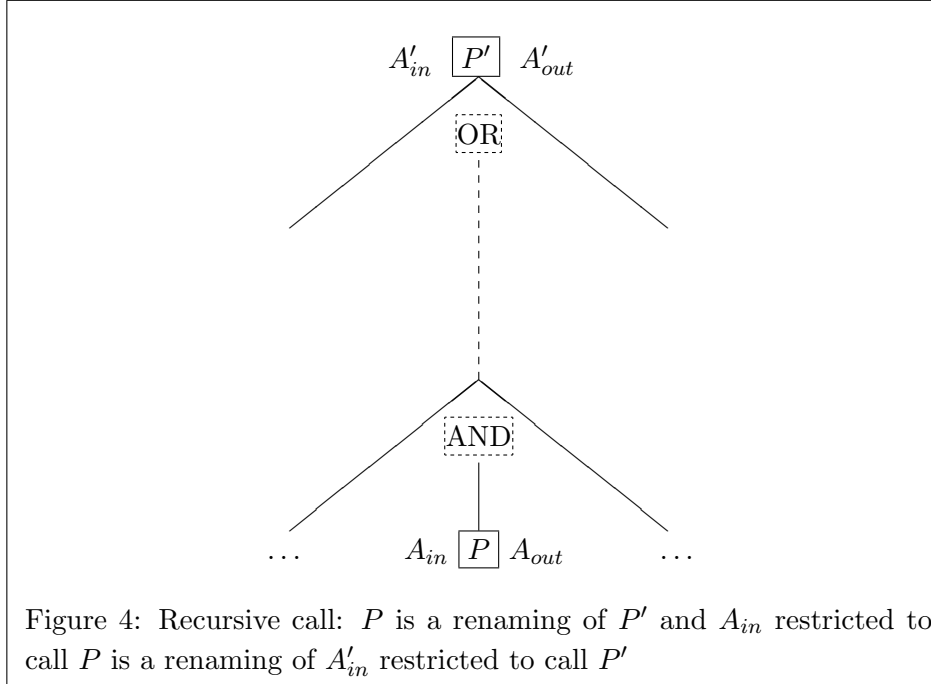


ancestor node is passed to this predicate call. Since the success abstraction of the predicate call may influence the success abstraction of the corresponding ancestor, the algorithm loops until these two abstractions are identical. The monotonicity property of all abstract operations together with the 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., $A' = ai-con(A, c)$ (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 abstraction⁶ (up to renaming of variables): First of all, we add an OR-node as shown in Figure 2 (H_1, \dots, H_m are the heads of all clauses for P). A_i^{in} is the abstraction computed by our abstract operations for the entry of clause $H_i :- \dots$ (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 $H :- L_1, \dots, L_k$ 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

⁶Recall 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).



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 [Bru91] 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 4.5, 4.6 and 4.7 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. We need the notion of “equivalence” of variables w.r.t. a constraint to formulate this theorem. Hence 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 $\sigma \in \text{Sol}(C)$.

Theorem 4.8 (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 $delay(Z'_1 \text{ or } Z'_2) \in A'$ with $Z_1 \sim_{C'} Z'_1$ and $Z_2 \sim_{C'} Z'_2$.*

Proof: We prove the theorem by induction on the depth of the concrete proof tree (AND-tree) for the literal L . Since all concrete proof trees are represented by the abstract rational AND-OR-tree computed by the abstract interpretation algorithm (cf. [Bru91]), we yield the proposition by comparing the concrete proof tree with the abstractions in the corresponding abstract tree.

For the base case we consider the possible single constraints for L :

- $L = X=Y$, $L = X=c$, $L = X=f(Y_1, \dots, Y_n)$, $L = X=Y_1+Y_2$, $L = X=Y_1-Y_2$, or $L = X \odot Y$ with $\odot \in \{<, >, <=, >=\}$: Since the evaluation of these constraints does not generate any new nonlinear constraints in C_L , the proposition is trivially true.
- $L = X=Y_1 * Y_2$: In this case we have $C_L = \{X=Y_1 * Y_2\}$. Since A is extended to $A' = A \cup \{Y_1, Y_2 \Rightarrow X, delay(Y_1 \text{ or } Y_2)\}$, this new nonlinear constraint is obviously covered by A' .

For the induction step we assume that $L = p(X_1, \dots, X_n)$ and $L_0 :- L_1, \dots, L_k$ is the clause selected for the proof of L . Then the root of the proof tree for L is marked with L and has proof trees for L_1, \dots, L_k as sons. The constraints C_0 present before L_1 is proved are $C_0 = C \cup \{L=L_0\}$. If A_0 is the abstraction computed from A and this clause in the corresponding abstract AND-OR-tree, then $C_0 \in \gamma(A_0)$ by Theorem 4.6. Thus we can apply our induction hypothesis to the proof tree for L_1 and we obtain that all nonlinear constraints in C_{L_1} (where C_{L_1} is the set of constraints added during execution of L_1) are covered by the success abstraction of L_1 . Moreover, Theorem 4.7 ensures that the constraints after execution of L_1 are contained in the concretisation of the corresponding abstraction. Therefore we can apply our induction hypothesis also to L_2 , and so on. Thus a simple induction on L_1, L_2, \dots, L_k yields the following result: If A_k is the success abstraction of L_k , then $delay \in A_k$ or for all $Z = Z_1 * Z_2 \in C_k$ (where C_k is the set of constraints added during execution of L_1, L_2, \dots, L_k , i.e., $C_L = \{L=L_0\} \cup C_k$) with $Z_1 * Z_2$ nonlinear in C' there exists $delay(Z'_1 \text{ or } Z'_2) \in A_k$ with $Z_1 \sim_{C'} Z'_1$ and $Z_2 \sim_{C'} Z'_2$. Hence we distinguish the following cases:

1. $delay \in A_k$: Then $delay \in A$ by definition of A in algorithm *ai*.
2. $delay(Z'_1 \text{ or } Z'_2) \in A_k$ with $Z'_1 \notin \text{var}(L_0)$ or $Z'_2 \notin \text{var}(L_0)$: Then $delay \in A$ by definition of A in algorithm *ai*.

3. $delay(Z'_1 \text{ or } Z'_2) \in A_k$ with $Z'_1, Z'_2 \in var(L_0)$: Let $L_0 = p(Y_1, \dots, Y_n)$. Then $Z'_1 = Y_i$ and $Z'_2 = Y_j$ for some $i, j \in \{1, \dots, n\}$. By definition of A in algorithm *ai*, $delay(X_i \text{ or } X_j) \in A$. Moreover, $X_i \sim_{C'} Y_i$ and $X_j \sim_{C'} Y_j$ because $p(X_1, \dots, X_n) = p(Y_1, \dots, Y_n) \in C'$. Thus $Z_1 \sim_{C'} X_i$ and $Z_2 \sim_{C'} X_j$ since $\sim_{C'}$ is an equivalence relation.

Altogether, the proposition is true in all cases. ■

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.

4.4 Example: Construction of an abstract AND-OR-tree

In the following we want to present an example which shows the construction of an abstract AND-OR-tree and the iteration over this tree by the abstract interpretation algorithm. The following flat $CLP(\mathcal{R})$ program, introduced in Section 3.3, relates a list of numbers with the product of all its elements:

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

The execution of this program may generate delayed nonlinear constraints if the first argument is not a list of numbers with unique values. For instance, the answer to the goal

```
?- prod([X,Y,Z], 10).
```

is “*maybe 10=X*Y*Z*”. We will show the computation of our abstract interpretation algorithm for such kind of goals. The initial goal is $prod(L, Pr)$ with abstraction $\{Pr\}$, i.e., the abstraction indicates that the second argument is a unique value while the first argument is arbitrary. The abstract interpretation algorithm applied to this goal and abstraction generates the abstract AND-OR-tree shown in Figure 5. This tree is constructed during the computation of the abstractions A_1, A_2, A_3 etc. It is finite since the rightmost leaf $prod(S, Q)$ has the same call abstraction as its ancestor OR-node $prod(R, P)$. In the following we present the results of the stepwise computations of abstractions in this tree:

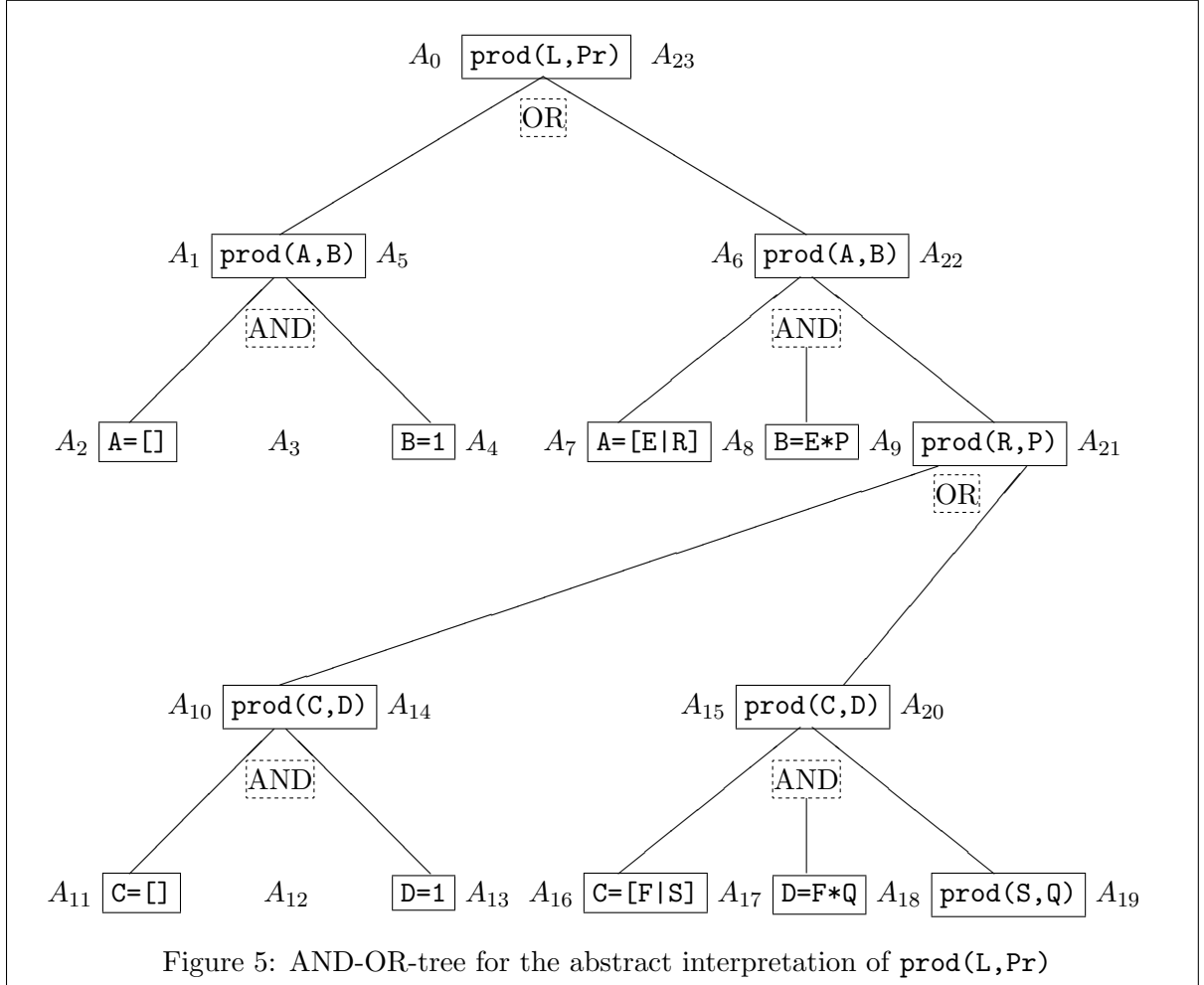
Initialization:

$$A_0 = \{Pr\}$$

Abstract interpretation of the first clause:

$$\begin{aligned} A_1 &= \{B\} \\ A_2 &= \{B\} \\ A_3 &= \{A, B\} \\ A_4 &= \{A, B\} \\ A_5 &= \{A, B\} \end{aligned}$$

Abstract interpretation of the second clause:



- $A_6 = \{B\}$
- $A_7 = \{B\}$
- $A_8 = \{B, \{E,R\} \Rightarrow A, \{A\} \Rightarrow E, \{A\} \Rightarrow R\}$
- $A_9 = \{B, \{E,R\} \Rightarrow A, \{A\} \Rightarrow E, \{A\} \Rightarrow R, \{E,P\} \Rightarrow B, \text{delay}(E \text{ or } P)\}$
 $\rightarrow \{B, \{E,R\} \Rightarrow A, \{A\} \Rightarrow E, \{A\} \Rightarrow R, \text{delay}(E \text{ or } P)\}$

The call abstraction part of A_9 is \emptyset which is different from A_0 . Hence it is not a recursive call and therefore a new OR-node is generated for the call `prod(R,P)`:

- $A_{10} = \emptyset$
- $A_{11} = \emptyset$
- $A_{12} = \{C\}$
- $A_{13} = \{C, D\}$
- $A_{14} = \{C, D\}$

Abstract interpretation of the second clause for `prod(R,P)`:

$$\begin{aligned}
A_{15} &= \emptyset \\
A_{16} &= \emptyset \\
A_{17} &= \{\{F, S\} \Rightarrow C, \{C\} \Rightarrow F, \{C\} \Rightarrow S\} \\
A_{18} &= \{\{F, S\} \Rightarrow C, \{C\} \Rightarrow F, \{C\} \Rightarrow S, \{F, Q\} \Rightarrow D, \text{delay}(F \text{ or } Q)\}
\end{aligned}$$

The call abstraction part of A_{18} and A_9 is \emptyset in both cases. Hence it is a recursive call. Instead of creating a new OR-node, the success abstraction of the ancestor call (A_{21}) is taken as a first success approximation of this recursive call. This is \perp because the success abstraction of the ancestor is currently not available:

$$\begin{aligned}
A_{19} &= \perp \\
A_{20} &= \perp \\
A_{21} &= A_9 \cup \{R, P\} \quad (\text{since } A_{14} \sqcup A_{20} = \{C, D\}) \\
&\quad \rightarrow^* \{B, R, P, \{E\} \Rightarrow A, \{A\} \Rightarrow E\}
\end{aligned}$$

The success abstraction of the ancestor node of the recursive call becomes available at this point. Hence we recompute the recursive call with the renamed second success approximation of $\text{prod}(R, P)$, i.e., with $\{S, Q\}$:

$$\begin{aligned}
A_{19} &= A_{18} \cup \{S, Q\} = \{S, Q, \{F, S\} \Rightarrow C, \{C\} \Rightarrow F, \{C\} \Rightarrow S, \{F, Q\} \Rightarrow D, \text{delay}(F \text{ or } Q)\} \\
&\quad \rightarrow^* \{S, Q, \{F\} \Rightarrow C, \{C\} \Rightarrow F, \{F\} \Rightarrow D\} \\
A_{20} &= \emptyset \\
A_{21} &= A_9 \cup \emptyset = A_9 \quad (\text{since } A_{14} \sqcup A_{20} = \emptyset)
\end{aligned}$$

The success abstraction of the ancestor node of the recursive call has changed. Hence we recompute the recursive call with the renamed third success approximation of $\text{prod}(R, P)$: \emptyset

$$\begin{aligned}
A_{19} &= A_{18} \cup \emptyset = \{\{F, S\} \Rightarrow C, \{C\} \Rightarrow F, \{C\} \Rightarrow S, \{F, Q\} \Rightarrow D, \text{delay}(F \text{ or } Q)\} \\
A_{20} &= \{\text{delay}\} \\
A_{21} &= A_9 \cup \{\text{delay}\} \quad (\text{since } A_{14} \sqcup A_{20} = \{\text{delay}\}) \\
&= \{B, \{E, R\} \Rightarrow A, \{A\} \Rightarrow E, \{A\} \Rightarrow R, \text{delay}(E \text{ or } P), \text{delay}\}
\end{aligned}$$

The success abstraction of the ancestor node of the recursive call has changed again. Fourth success approximation of the recursive call $\text{prod}(R, P)$: $\{\text{delay}\}$

$$\begin{aligned}
A_{19} &= A_{18} \cup \{\text{delay}\} = \{\{F, S\} \Rightarrow C, \{C\} \Rightarrow F, \{C\} \Rightarrow S, \{F, Q\} \Rightarrow D, \text{delay}(F \text{ or } Q), \text{delay}\} \\
A_{20} &= \{\text{delay}\} \\
A_{21} &= A_9 \cup \{\text{delay}\} \quad (\text{since } A_{14} \sqcup A_{20} = \{\text{delay}\}) \\
&= \{B, \{E, R\} \Rightarrow A, \{A\} \Rightarrow E, \{A\} \Rightarrow R, \text{delay}(E \text{ or } P), \text{delay}\}
\end{aligned}$$

No further iteration is necessary since the success abstraction of $\text{prod}(R, P)$ has not changed:

$$\begin{aligned}
A_{22} &= \{B, \text{delay}\} \\
A_{23} &= \{\text{Pr}, \text{delay}\} \quad (\text{since } A_5 \sqcup A_{22} = \{B, \text{delay}\})
\end{aligned}$$

The element *delay* in the success abstraction of the initial goal indicates that the concrete computation may stop with a delayed nonlinear constraint.

5 Extension to other delayed constraints

In Section 2 we have defined the subclass of $\text{CLP}(\mathcal{R})$ programs which we can analyse by our abstract interpretation algorithm. However, $\text{CLP}(\mathcal{R})$ 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 $Z=\text{sin}(X)$ is delayed until X is ground while the constraint $Z=\text{abs}(X)$ is delayed until X is ground, $Z=0$ or Z is ground and negative [HJM⁺91]. Since the exact value of a ground variable is not available in our abstract domain, we can only approximate this behaviour 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.

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

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

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

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

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

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

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

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

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 $\text{CLP}(\mathcal{R})$ program. In this section we will outline possible applications of this algorithm.

6.1 Better user support

In $\text{CLP}(\mathcal{R})$ 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 $\text{CLP}(\mathcal{R})$ 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 4.4). 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 $\text{CLP}(\mathcal{R})$ constraint solver is able to decide 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, then nonlinear constraints cannot occur at run time. Hence the program can be compiled without the delay mechanism for nonlinear constraints [JMY91]. Moreover, general instructions for creating nonlinear constraints can be specialized to simpler instructions for creating linear constraints [JMSY92a].
2. The RISC-CLP(Real) system [Hon92] allows the formulation of nonlinear arithmetic constraints which are not delayed but checked by a powerful constraint solver. But 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

⁷For 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.

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 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 [JL87]. This allows an early failure detection and avoids infinite derivation paths which may be present in pure logic programming. However, in $\text{CLP}(\mathcal{R})$ this advantage is sometimes lost since nonlinear constraints are not checked for satisfiability. For instance, consider the following $\text{CLP}(\mathcal{R})$ 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 $\text{?-fac}(8,F)$ 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 $\text{?-fac}(N,24)$. In this case $\text{CLP}(\mathcal{R})$ 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 $\text{?-fac}(N,10)$, $\text{CLP}(\mathcal{R})$ 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 $\text{CLP}(\mathcal{R})$ due to the delay mechanism. If we use a more powerful constraint solver which is able to treat nonlinear constraints (like in CAL [ASS⁺88] or RISC-CLP(Real) [Hon92]), 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 $\text{CLP}(\mathcal{R})$ 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 $\text{CLP}(\mathcal{R})$ 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 [Bru91] 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 [MS90] 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, [Mel85, Nil88, Bru91] or [JMM91] 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” [JMSY92b, VH89] 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 [JMM91] 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 $\{X\} \Rightarrow NX$, $\{NX,Y\} \Rightarrow NXY$ and $\{Z\} \Rightarrow NZ$ (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 [MSD90, CFW91]. 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 can 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 [Han92]. 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 [GdlBH92] 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. The author is grateful to Peter Barth and Frank Zartmann for discussions and their careful reading of this paper.

References

- [AH87] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.
- [ASS⁺88] 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.
- [Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming* (10), pp. 91–124, 1991.
- [CC77] 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.
- [CFW91] 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.
- [Col90] A. Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, Vol. 33, No. 7, pp. 69–90, 1990.
- [Deb89] 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.
- [GdlBH92] 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.
- [Han91] M. Hanus. Parametric Order-Sorted Types in Logic Programming. In *Proc. of the TAPSOFT '91*, pp. 181–200. Springer LNCS 494, 1991.
- [Han92] 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.
- [HJM⁺91] 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.
- [Hon92] 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.
- [JL87] 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.
- [JMM91] 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.
- [JMSY92a] 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.
- [JMSY92b] 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.

- [JMY91] 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.
- [LCMVH91] 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.
- [Mel85] C.S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming (1)*, pp. 43–66, 1985.
- [MS90] 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.
- [MSD90] 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.
- [Nil88] 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.
- [Nil90] 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.
- [Pfe92] F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.
- [VH89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.