Freeness Analysis in the Presence of Numerical Constraints

Veroniek Dumortier, Gerda Janssens,
Maurice Bruynooghe, Michael Codish
K.U. Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
veroniek@cs.kuleuven.ac.be

Abstract

This paper describes a novel application of abstract interpretation dealing with mixed systems of numerical and unification constraints. In general, the abstraction of such systems cannot be separated into two independent parts, since a program variable is often involved in both types of constraints. The proposed abstraction deals in an elegant way with the interaction between the two types of constraints. The abstract domain extends the notion of possible sharing information in the context of logic programs towards possible dependencies established by constraint systems. The usefulness of the information on possible dependencies between program variables is twofold: (1) it is used to derive definite freeness information needed for low-level optimisations; (2) it also allows to delay the addition of constraints in a computation – i.e. high-level optimisation – thus augmenting the opportunity for constraint specialisation. Coupled with results from other analyses, the derived information is highly valuable in the compilation of CLP programs to reduce the overhead of the general constraint solver.

1 Introduction

The family of constraint logic programming (CLP) languages [12, 4, 5] substantially increases the expressiveness and declarativeness of logic programming languages. However, applying general constraint solving algorithms is expensive and should be avoided where possible. Experiments with current CLP systems show that a substantial speed-up is obtained by specialising constraints at compile-time (e.g. by translating constraints into tests or assignments) [15]. The analysis described in this paper provides information essential to perform such transformations.

The basic philosophy which led to the introduction of constraint logic languages views unification — the core of logic programming — as imposing constraints over the Herbrand universe. It is then meaningful to generalise this concept and to consider constraints over other domains such as the real numbers (CLP($\mathcal{R}$) [13]) or the rational numbers (Prolog III [5]).

Constraint logic programs consist of clauses of the form $h \leftarrow \{\mathcal{C}\},\ \mathcal{B}$ where h is an atom called the *head*, $\mathcal{C}$ is a (possibly empty) conjunction of constraints, and $\mathcal{B}$ is a (possibly empty) sequence of atoms called the body. A query is of the form $\leftarrow \{\mathcal{C}\},\ \mathcal{B}$ where $\mathcal{C}$ is a (possibly empty) conjunction

of constraints, sometimes called the *current store*, and $\mathcal{B}$ is a sequence of atoms. The basic step in a CLP computation is similar to that of logic programs. However, instead of unification the more general notion of *constraint satisfiability* is applied. The reduction of a query $\leftarrow \{\mathcal{C}\},\ b_1, \ldots, b_n$ with a (renamed) program clause $h \leftarrow \{\mathcal{C}_c\},\ \mathcal{B}_c$ is enabled if the constraint system $\mathcal{C}' = (\mathcal{C} \wedge \{h = b_1\} \wedge \mathcal{C}_c)$ is satisfiable. In this case, the query is rewritten into the new query $\leftarrow \{\mathcal{C}'\},\ \mathcal{B}_c :: \langle b_2, \ldots, b_n \rangle$, where :: denotes concatenation of sequences. Logic programming is a special case in which constraints are only unification constraints. In this case checking for constraint satisfiability coincides with finding a most general unifier, for which there exist efficient algorithms. In contrast, for more general constraints such as linear equations and inequalities, the algorithms for checking satisfiability of the constraint store each time a new constraint is added are more expensive. Hence, it is advantageous to identify cases in which constraints are trivial, i.e. degrade into a simple test or assignment.

The framework of abstract interpretation [6] provides the basis for a semantic approach to dataflow analysis. In this approach a program analysis is viewed as a non-standard, abstract semantics defined over a domain of data descriptions. An abstract semantics is constructed by replacing operations in a suitable concrete semantics with corresponding abstract operations defined on data descriptions. Program analyses are obtained by providing finitely computable abstract interpretations which preserve interesting aspects of program behaviour. In the case of logic programs, *abstract substitutions* typically specify sets of substitutions which describe the call and success patterns of a given clause and *abstract unification* mimics unifications involving such descriptions.

The use of abstract interpretation for CLP has been discussed in [20] by Marriott and Sondergaard. Adapting the framework of [1] for LP to the analysis of CLP is described in [10]. Low-level optimisations based on abstract interpretation have been discussed by several authors [20, 15, 14]. In a recent paper, Marriott and Stuckey [21] discuss three global optimisations: constraint refinement, constraint removal and constraint reordering using a convex hull approximation [9] of the solution space of a system of linear equations and inequalities, and an abstract Fourier algorithm to approximate the satisfiability property of such systems.

The contribution of this paper is twofold:

- It introduces an abstraction for constraint systems containing numerical and unification constraints. In general, such systems cannot be separated into two independent parts, since it frequently happens that a program variable is involved in both numerical and unification constraints. This problem has not been addressed before. The abstraction keeps track of possible dependencies between program variables and allows to derive freeness information.
- It presents a condition for reordering constraints in clause bodies which is based on our abstraction. This condition is simple and easy to verify.

Example 1.1
Consider the following constraint logic program and a query $\leftarrow \{\mathcal{C}\}$, $sumlist(L0, S0)$ *in which* $\mathcal{C}$ *constrains the first argument to be a ground list* $L0$ *and the second argument is a free variable* $S0$.

$sumlist(L, S) \leftarrow \{L = [X \mid T], S = S' + X\},\ sumlist(T, S').$
$sumlist(L, S) \leftarrow \{L = [\,], S = 0\}.$

A reduction of such a query with the recursive clause invokes a constraint $S = S' + X$. *Here,* S *and* S' *are free variables and* X *is ground since it is also involved in the unification constraint* $L = [X \mid T]$. *Due to the freeness of both* S *and* S', *the constraint* $S = S' + X$ *is nontrivial in the sense that it cannot be compiled to a test or to an assignment . Moreover, since* $S = S' + X$ *may interact with other constraints in* $\mathcal{C}$, *the second argument in the recursive call can no longer be determined free. Hence the technique of [15] adds the constraint to the store and offers no optimisation.*

In contrast, if global analysis infers that $S = S' + X$ *does not interact with* $\mathcal{C}$, *then the fact that* S *and* S' *are free implies that* $S = S' + X$ *cannot bind* S *or* S'. *Consequently, the recursive call has the original call pattern (i.e., the first argument is ground and the second is free). Moreover,* S *does not participate in the call (directly nor indirectly through sharing), so it is safe to delay adding the constraint* $S = S' + X$ *until after returning from the recursive call. Interestingly enough, at that point* S' *is ground (due to the base case), so the constraint becomes trivial and can be translated into the assignment* $S := S' + X$.

Remark that it is not only the compilation of the numerical constraint satisfaction that benefits from the freeness information of variables, but as these variables can occur in unifications, also the optimisations based on the modes **ground, free** *and* **any** *for logic programs can be used [19, 26, 27].*

The structure of the paper is as follows. Section 2 defines some preliminary notions. Section 3 introduces the abstract domain and defines the abstraction of systems of linear equations, systems of unifications and mixed systems involving both types of constraints. Section 4 defines the basic abstract operations – abstract conjunction and projection. Section 5 describes the other abstract operations and the treatment of more general numerical constraints. The results obtained by the analysis are evaluated and the integration of the analysis into a full mode analysis system is discussed.

2 Preliminaries

In the following we assume familiarity with the standard definitions and notation for logic programs [18] and abstract interpretation [6]. We assume a set of function symbols $\Sigma = \Sigma_N \cup \Sigma_A$ (Σ_N and Σ_A overlap) and a denumerable set of variables Var. Function symbols are classified as *numerical* or *alphabetical*, i.e. in Σ_N or Σ_A respectively. The numerical function symbols will typically include: *numerical constants* such as real, integer or rational

numbers — depending on the specific domain; and *numerical functors* such as $+, -, \times$ and $/$. The alphabetical function symbols include the constants (including numbers) and non-numerical function symbols. The corresponding term algebras are denoted respectively $T(\Sigma_N, Var)$ and $T(\Sigma_A, Var)$.

A *numerical constraint* is an equation, disequation or inequality between $T(\Sigma_N, Var)$ terms. In the first part of this paper we focus on numerical constraints which are linear equations written as $a_1X_1 + \ldots + a_nX_n = b$ (where $a_1, \ldots, a_n, b$ are numbers). In Section 5 we discuss the implications of our results for general numerical constraints. A *unification constraint* is an equation between $T(\Sigma_A, Var)$ terms. A *constraint system* is a finite conjunction of numerical and unification constraints, sometimes written as a set. Note that while an equation of the form $X = f(Y + 1, 2 * Z - 3, T)$ is neither a numerical nor a unification constraint, it can be *normalised* into the system $\{X = f(A, B, T), A - Y = 1, B - 2Z = 3\}$ consisting of a unification and two numerical constraints.

It is well known that (finite) conjunctions of (solvable) unification constraints are isomorphic to the class of idempotent substitutions [16]. Hence, we do not distinguish them in the following. We denote by $s\theta$ the application of the substitution $mgu(\theta)$[1] to syntactic object s.

The set of all linear combinations of a finite conjunction of (solvable) linear equations E is denoted $lc(E)$[2].

Let $\mathcal{C}$ be a constraint system. It is possible that the (numerical) constraints in $\mathcal{C}$ entail a unification constraint. Likewise, the (unification) constraints in $\mathcal{C}$ may entail a numerical constraint. We denote by $num^*(\mathcal{C})$ and $unif^*(\mathcal{C})$ the respective conjunctions of all numerical and unification constraints entailed by $\mathcal{C}$. Note that $unif^*(\mathcal{C})$ always has a finite solved form and hence is represented by its most general unifier.

Example 2.1
For $\mathcal{C} = \{X = f(Y), X = f(Z), Y - Z - T + U = 0, W = X\}$,
$num^*(\mathcal{C}) = lc(\{Y - Z - T + U = 0, Y - Z = 0, T - U = 0\})$ *and*
$unif^*(\mathcal{C}) = \{Y = Z, T = U, X = f(Z), W = f(Z)\}$.

A variable X occurring in a constraint system $\mathcal{C}$ is classified as *non-numerical* if $\mathcal{C}$ entails $X = t$ with $t \notin T(\Sigma_N, Var)$. A constraint system is called *well-typed* if non-numerical variables do not occur in numerical constraints. For example, $\mathcal{C} = \{X = Y, Y = f(Z), X + 3 = T\}$ is not well-typed since X is non-numerical ($\mathcal{C}$ entails $X = f(Z)$) and occurs in the constraint $X + 3 = T$. A variable X that is not non-numerical, however, may occur in both $num^*(\mathcal{C})$ and $unif^*(\mathcal{C})$, e.g. this is the case with T and U in example 2.1 ($T - U = 0 \in num^*(\mathcal{C})$, $T = U \in unif^*(\mathcal{C})$). In the sequel, we focus on the analysis of constraint logic programs containing *well-typed normalised* constraint systems.

[1] mgu is a partial function which maps a conjunction of unification constraints to a *unique* most general unifier if it exists.

[2] The definition of $lc(E)$ is assumed to exclude equations of the form $b = b$.

For simplicity, we assume the standard framework of abstract interpretation as defined in [6]. Our concrete domain, denoted $(Con_{\mathcal{C}}, \subseteq)$, consists of *sets of constraint systems*. The abstract domain, $(Con_{\mathcal{A}}, \subseteq)$, consists of *abstract constraint systems* which are sets of sets of variables. The least upper bound of two or more abstract constraint systems is their union.

Abstraction is formalised in terms of a Galois insertion $(Con_{\mathcal{C}}, \alpha, Con_{\mathcal{A}}, \gamma)$[3]. In the following abstractions are defined in terms of the abstraction function α leaving the corresponding concretisation function determined as described in [6].

Definition 2.1 (Abstraction of a set of constraint systems)
Let $\mathcal{E}$ be a set of constraints systems. Then $\alpha(\mathcal{E}) = \cup\{\alpha(\{\mathcal{C}\}) \mid \mathcal{C} \in \mathcal{E}\}$.
(by abuse of notation we often write $\alpha(\mathcal{C})$ instead of $\alpha(\{\mathcal{C}\})$)

Stated pragmatically, the abstract interpretation of constraint logic programming requires the introduction of two abstract operations:

- abstract conjunction: to enable the addition of a constraint system to an abstract constraint system;
- abstract projection: to enable the projection of an abstract constraint system onto a given set of variables.

Other operations, such as *procedure-entry* and *procedure-exit* as required in the framework of [1], are easily expressed in terms of the above operations.

3 Abstract domain

In the context of logic programs, it is well known that freeness analysis is more precise when coupled with sharing analysis e.g. as in [24]. This is because when a variable X becomes further instantiated any free variable which shares with X may potentially become non-free. Consequently, if sharing information is not available then all variables become potentially non-free. We illustrate here that the same holds for linear equations and for systems containing both unifications and linear equations.

Intuitively, a variable X is free in a constraint system if there is no equivalent constraint system which contains an equation $X = t$ where t is a non-variable term. Consider a system of linear equations E. Equivalent systems are obtained by considering linear combinations of the equations in E. A variable X is free if no such combination gives an equation of the form $X = n$ where n is a number. For example, in $\{X + Y - Z = 0, X - Y = 0\}$, the variables X, Y and Z are free. However adding the constraint $X = 1$ results in a system which has $X = 1$, $Y = 1$ and $Z = 2$ as linear combinations implying that X, Y and Z are non-free. Our abstraction will provide definite freeness information by approximating the set of all linear combinations of a given system of linear equations.

[3] I.e., $\alpha : Con_{\mathcal{C}} \to Con_{\mathcal{A}}$ and $\gamma : Con_{\mathcal{A}} \to Con_{\mathcal{C}}$ are monotonic *abstraction* and *concretisation* functions, $\alpha(\gamma(\mathcal{AC})) = \mathcal{AC}$ and $\mathcal{C} \subseteq \gamma(\alpha(\mathcal{C}))$ for $\mathcal{C} \in Con_{\mathcal{C}}$ and $\mathcal{AC} \in Con_{\mathcal{A}}$.

For unification constraints, non-freeness is also detected by considering combinations of constraints, this time as derived by applications of Clark's equality theory [2]. A variable X is non-free in a system of unification constraints if it is equal (under this theory) to a non-variable term.

More complex, is the analysis of mixed constraint systems where both types of combinations should be applied. For example, consider the constraint system $\{X = f(Y), Z + Y - W = 0, Z = 0\}$. The linear combinations of the numerical constraints indicate that Y and W are free and $Y = W$. Combining unification constraints does not bind Y and W to non-variable terms, hence the freeness is maintained.

We introduce an abstraction which describes possible variable dependencies established by mixed systems of linear equations and unification constraints. These dependencies are useful not only for deriving freeness information but also for detecting cases in which it is safe to delay the addition of a constraint to the store. A constraint may safely be "shifted" over a call b if it does not (together with the current store) establish a dependency between (a subset of) the variables in b or constrain a variable in b to be possibly non-free.

3.1 Abstracting linear equations

A *system of linear equations* E has the form :

$$\begin{array}{llllll} a_{11}X_1 & + & a_{12}X_2 & +\ldots+ & a_{1n}X_n & = b_1 \\ \ldots\ldots & & \ldots\ldots & & \ldots\ldots & \ldots\ldots \\ a_{m1}X_1 & + & a_{m2}X_2 & +\ldots+ & a_{mn}X_n & = b_m \end{array}$$

where $X_1, \ldots, X_n$ are variables and a_{ij} and b_i $(i = 1..m,\ j = 1..n)$ are numbers. We say that E is solvable, denoted by $solvable(E)$, if there exists a *solution* $(c_1, c_2, \ldots, c_n)$ with c_i $(i = 1..n)$ being numbers such that for each equation $(j = 1..m)$ $a_{j1}c_1 + a_{j2}c_2 \ldots + a_{jn}c_n = b_j$. We say that two systems of linear equations are *equivalent* if they have the same solutions. Given a system E, we are interested in finding equivalent systems containing equations of the form $X = n$. These may be obtained by considering linear combinations of the equations in E, denoted $lc(E)$.

Definition 3.1 (Abstracting a system of linear equations) *Let E be a solvable system of linear equations. Then*

$$\alpha(E) = \left\{ \{X_1, \ldots, X_n\} \;\middle|\; \begin{array}{l} (a_1X_1 + \cdots + a_nX_n = b) \in lc(E), \\ a_i \neq 0 \ (i = 1..n) \end{array} \right\}.$$

We note that although $lc(E)$ is infinite, $\alpha(E)$ is finite and computable as only finitely many linear combinations need to be considered.

To motivate this definition observe that : (1) if $\{X\} \notin \alpha(E)$ then X is definitely free in E; and (2) if for all S such that $S \subseteq \{X_1, \ldots, X_n\}$ and S is not a singleton it holds that $S \notin \alpha(E)$, then the variables $\{X_1, \ldots, X_n\}$ are

independent, i.e. further instantiation of any of the variables in $\{X_1, \ldots, X_n\}$ does not effect the freeness of the others. This intuition will be formalised in Proposition 3.1 (below).

Example 3.1
For $E = \{X + Y = 4, Y + Z = 3\}$, $\alpha(E) = \{\{X, Y\}, \{Y, Z\}, \{X, Y, Z\}, \{X, Z\}\}$ *. The abstraction indicates that* X, Y *and* Z *are free and that further instantiation of any variable in* E *may cause* X, Y *and* Z *to become non-free.*

3.2 Abstracting unification constraints

For systems of unification constraints, variable dependencies are also detected by considering all possible combinations of constraints. For example, if $\theta = \{f(X, Y) = f(a, g(B)), Z = B\}$, then X and Y are non-free because $X = a$ and $Y = g(B)$ are entailed by θ. Moreover there is a dependency between Y and Z because of the shared variable B. Thus, if Y becomes further instantiated then also Z may become non-free.

Definition 3.2 (Abstracting a system of unification constraints)
Let θ *be a solvable system of unification constraints and let*

$$\begin{aligned} H \;=\; & \left\{ \{X\} \;\middle|\; X\theta \text{ is a non-variable} \right\} \cup \\ & \left\{ \{X, Y\} \;\middle|\; X \not\equiv Y, vars(X\theta) \cap vars(Y\theta) \neq \phi \right\}. \end{aligned}$$

Then, $\alpha(\theta) = \uplus H$ *where* $\uplus H$ *denotes the closure of* H *under union.*

The motivation of this definition is similar to that of Definition 3.1.

Example 3.2
Let $\theta_1 = \{X = f(A, B), Y = g(A)\}$ *and* $\theta_2 = \{X = f(C, D), Y = g(C), C = 3\}$*.* θ_1 *is in solved form and* $mgu(\theta_2) = \{X = f(3, D), Y = g(3), C = 3\}$*.*

$$\begin{aligned} \alpha(\theta_1) \;&=\; \uplus\{\; \{X\}, \{Y\}, \{X, A\}, \{X, B\}, \{X, Y\}, \{A, Y\} \;\} \\ &=\; \left\{ \begin{array}{l} \{X\}, \{Y\}, \{X, A\}, \{X, B\}, \{X, Y\}, \{A, Y\}, \\ \{X, A, Y\}, \{X, B, Y\}, \{X, A, B\}, \{X, A, B, Y\} \end{array} \right\}. \end{aligned}$$

Hence, A *and* B *are free and independent;* X *and* A*, as well as* X *and* B*, are not independent, so further instantiation of* X *may cause* A *and* B *to become non-free; also,* Y *and* B *are independent while* Y *and* A *are not.*

$$\begin{aligned} \alpha(\theta_2) \;&=\; \uplus\{\; \{X\}, \{Y\}, \{C\}, \{X, D\} \;\} \\ &=\; \left\{ \begin{array}{l} \{X\}, \{Y\}, \{C\}, \{X, D\}, \{X, C\}, \{X, Y\}, \{Y, C\}, \\ \{X, C, D\}, \{X, Y, D\}, \{X, Y, C\}, \{X, Y, C, D\} \end{array} \right\}. \end{aligned}$$

Hence, D *is the only free variable and is dependent on* X*.*

Note that taking the closure under union in the definition of α does not introduce a loss of precision. Freeness and independence of variables are determined from the "smallest" sets in a description. Also note that not all descriptions are closed under union. In particular in Example 3.2, $\alpha(\{\theta_1, \theta_2\}) = \alpha(\theta_1) \cup \alpha(\theta_2)$ (by Definition 2.1) is not.

3.3 Abstracting mixed constraint systems

As mentioned above, when dealing with mixed constraint systems we must consider both linear combinations and manipulation of unification constraints.

Example 3.3
Consider the constraint system $\mathcal{C} = \{X = f(A), A + B = 3\}$. $X = f(A)$ *establishes a dependency between* X *and* A *and* $A + B = 3$ *creates a dependency between* A *and* B. *However, there is an additional dependency between* X *and* B *due to the entailed equation* $X = f(3 - B)$. *Thus, if* X *becomes further instantiated then also* B *may become non-free.*

Definition 3.3 (Abstracting a mixed constraint system)
Let $\mathcal{C}$ *be a solvable mixed constraint system, let* $\theta = unif^*(\mathcal{C})$ *and let*

$$
\begin{aligned}
H \;=\; & \left\{ \{X\} \;\middle|\; X\theta \text{ is a non-variable} \right\} \\
& \cup \left\{ \{X, Y\} \;\middle|\; X \not\equiv Y,\ vars(X\theta) \cap vars(Y\theta) \neq \phi \right\} \\
& \cup \left\{ \{X_1, \ldots, X_n\} \;\middle|\; \begin{array}{l} (a_1X_1 + \ldots + a_nX_n = b) \in num^*(\mathcal{C}), \\ a_i \neq 0 \ (i = 1..n) \end{array} \right\} \\
& \cup \left\{ \begin{array}{l} \{Y_1, \ldots, Y_m, X_1, \ldots, X_n\} \,|\, Y_i\theta \text{ is a compound term}, \\ Z_i \in vars(Y_i\theta), \text{all } Z_i \text{ are distinct}, \\ (b_1Z_1 + \ldots + b_mZ_m + a_1X_1 + \ldots + a_nX_n = b) \\ \in num^*(\mathcal{C}), b_i \neq 0, a_j \neq 0 \ (i = 1..m, j = 1..n) \end{array} \right\}
\end{aligned}
$$

Then $\alpha(\mathcal{C}) = \uplus H$ *where* $\uplus H$ *denotes the closure of* H *under union.*

Note that although $num^*(\mathcal{C})$ may be infinite, only a finite number of linear combinations of equations need to be considered to compute $\alpha(\mathcal{C})$. So, $\alpha(\mathcal{C})$ is finite and computable.

The first set in the definition of H yields information on the non-freeness of variables (variables bound to a ground or compound term). The second and third set describe variable dependencies that are established through all possible entailed unification constraints, resp. through entailed linear equations. The last set contains variable dependencies established by combining unification constraints and numerical constraints. It indicates dependencies between non-numerical and numerical variables.

Example 3.4
Let $\mathcal{C} = \{X = f(A, B), A + B + T = 3\}$. *Then,*

$$
\begin{aligned}
unif^*(\mathcal{C}) &= \{X = f(A, B)\}, \\
num^*(\mathcal{C}) &= lc(\{A + B + T = 3\}), \\
H &= \left\{ \begin{array}{l} \{X\}, \{X, A\}, \{X, B\}, \{A, B, T\}, \{X, B, T\}, \\ \{X, A, T\}, \{X, T\} \end{array} \right\}, \\
\alpha(\mathcal{C}) &= \left\{ \begin{array}{l} \{X\}, \{X, A\}, \{X, B\}, \{A, B, T\}, \{X, B, T\}, \\ \{X, A, T\}, \{X, T\}, \{X, A, B\}, \{X, A, B, T\} \end{array} \right\}.
\end{aligned}
$$

Our notion of abstraction is motivated by the following:

Proposition 3.1
Let $\mathcal{AC} \in Con_{\mathcal{A}}$ and $\mathcal{C} \in Con_{\mathcal{C}}$ such that $\alpha(\mathcal{C}) \subseteq \mathcal{AC}$. Then: (1) $\{X\} \notin \mathcal{AC}$ implies that X is free in $\mathcal{C}$; and (2) if for all S, such that $S \subseteq \{X_1, \ldots, X_n\}$ and S is not a singleton, it holds that $S \notin \mathcal{AC}$, then $\{X_1, \ldots, X_n\}$ are independent, i.e. instantiating any of the X_i cannot trigger the instantiation of the other variables in the set.

4 Abstracting conjunction and projection

The basic operation of a CLP system is the conjunction of a new constraint (system) to the current constraint store. We introduce a corresponding abstract conjunction which mimics this operation.

Let $\mathcal{AC}_1$ and $\mathcal{AC}_2$ be abstract constraint systems. The abstract conjunction of $\mathcal{AC}_1$ and $\mathcal{AC}_2$ is denoted $\mathcal{AC}_1 \wedge \mathcal{AC}_2$. The effect of adding the constraint system $\mathcal{C}$ to an abstract constraint store $\mathcal{AC}$ is computed by $\mathcal{AC} \wedge \alpha(\mathcal{C})$.

Definition 4.1 (Abstract conjunction)
The abstract conjunction of $\mathcal{AC}_1, \mathcal{AC}_2 \in Con_{\mathcal{A}}$ is defined by

$$\mathcal{AC}_1 \wedge \mathcal{AC}_2 = \mathcal{AC}_1 \cup \mathcal{AC}_2 \cup ((\mathcal{AC}_1 \oplus \mathcal{AC}_2) \setminus \{\phi\}) \qquad \textit{where}$$

$$\mathcal{AC}_1 \oplus \mathcal{AC}_2 = \left\{ (A_1 \cup A_2) \setminus D \;\middle|\; A_1 \in \mathcal{AC}_1, A_2 \in \mathcal{AC}_2, D \subseteq A_1 \cap A_2 \right\}.$$

Example 4.1
Let $\mathcal{AC} = \{\{X, T\}\}$ and $\mathcal{C} = \{X + T = Z, Y = f(Z)\}$.

$$\begin{aligned} \alpha(\mathcal{C}) &= \{ \{Y\}, \{Y, Z\}, \{X, T, Z\}, \{X, T, Y\}, \{X, T, Y, Z\} \}. \\ \mathcal{AC} \wedge \alpha(\mathcal{C}) &= \left\{ \begin{array}{l} \{X, T\}, \{Y\}, \{Y, Z\}, \{X, T, Z\}, \{X, T, Y\}, \\ \{X, T, Y, Z\}, \{X, Z\}, \{T, Z\}, \{Z\}, \{X, Y\}, \\ \{T, Y\}, \{X, Y, Z\}, \{T, Y, Z\} \end{array} \right\}. \end{aligned}$$

After conjunction, Z and Y are possibly non-free, while T and X are definitely free.

Proposition 4.1
Let $\mathcal{C}_1, \mathcal{C}_2 \in Con_{\mathcal{C}}$ and $\mathcal{AC}_1, \mathcal{AC}_2 \in Con_{\mathcal{A}}$. Then

$$\alpha(\mathcal{C}_1) \subseteq \mathcal{AC}_1 \quad \wedge \quad \alpha(\mathcal{C}_2) \subseteq \mathcal{AC}_2 \quad \Rightarrow \quad \alpha(\mathcal{C}_1 \wedge \mathcal{C}_2) \subseteq \mathcal{AC}_1 \wedge \mathcal{AC}_2.$$

Sketch of the proof

Although $\mathcal{AC}_1$ and $\mathcal{AC}_2$ are not necessarily closed under union, there are subsets $\mathcal{AC}'_1 \subseteq \mathcal{AC}_1$ and $\mathcal{AC}'_2 \subseteq \mathcal{AC}_2$ such that $\alpha(\mathcal{C}_1) = \mathcal{AC}'_1$ and $\alpha(\mathcal{C}_2) = \mathcal{AC}'_2$. Every equation in $unif^*(\mathcal{C}_1 \wedge \mathcal{C}_2)$ and in $num^*(\mathcal{C}_1 \wedge \mathcal{C}_2)$ is abstracted in $\mathcal{AC}'_1 \wedge \mathcal{AC}'_2 \subseteq \mathcal{AC}_1 \wedge \mathcal{AC}_2$, so $\alpha(\mathcal{C}_1 \wedge \mathcal{C}_2) \subseteq \mathcal{AC}_1 \wedge \mathcal{AC}_2$. (This proof motivates the use of the closure under union in definition 3.2 and 3.3).

The other basic operation is projection. In the concrete domain, projection can be defined as follows:

Definition 4.2 (Concrete projection)
Given a constraint system $\mathcal{C}$, the projection of $\mathcal{C}$ on $V \subseteq Var$ is defined as

$$\mathcal{C} \mid_V = \{c \mid vars(c) \subseteq V, c \in num^*(\mathcal{C}) \cup unif^*(\mathcal{C})\}$$

This projection has a number of interesting properties:

Proposition 4.2
Let $\mathcal{C} \in Con_{\mathcal{C}}$ and $V \subseteq Var$. Then,

1. $\forall\ \mathcal{C}' \in Con_{\mathcal{C}}$ over V : $\mathcal{C} \models \mathcal{C}'$ iff $\mathcal{C} \mid_V \models \mathcal{C}'$;

2. $\forall\ \mathcal{C}', \mathcal{C}'' \in Con_{\mathcal{C}}$ over V : $(\mathcal{C} \wedge \mathcal{C}') \models \mathcal{C}''$ iff $(\mathcal{C} \mid_V \wedge \mathcal{C}') \models \mathcal{C}''$.

The proposition expresses that projection does not cause loss of information. If the computation is in a state where only constraints over V are to be added, then no pruning will be lost when the store is projected onto the variables of V and the computation is continued with the reduced store. The existence of projections with these properties has to do with the inherent algebraic properties of the constraint domain (cfr. [11] for more details).

Definition 4.3 (Abstract projection)
Given an abstract constraint system $\mathcal{AC}$, the abstract projection of $\mathcal{AC}$ on $V \subseteq Var$ is defined as

$$\mathcal{AC} \mid_V = \{S \in \mathcal{AC} \mid S \subseteq V\}$$

Proposition 4.3
Let $\mathcal{C} \in Con_{\mathcal{C}}$, $\mathcal{AC} \in Con_{\mathcal{A}}$ such that $\alpha(\mathcal{C}) \subseteq \mathcal{AC}$. Then $\alpha(\mathcal{C} \mid_V) \subseteq \mathcal{AC} \mid_V$.

Proposition 4.4
Let $\mathcal{AC} \in Con_{\mathcal{A}}$ and $V \subseteq Var$. Then

1. $\forall\ \mathcal{AC}' \in Con_{\mathcal{A}}$ over V : $\mathcal{AC}' \subseteq \mathcal{AC}$ iff $\mathcal{AC}' \subseteq \mathcal{AC} \mid_V$;

2. $\forall\ \mathcal{AC}', \mathcal{AC}'' \in Con_{\mathcal{A}}$ over V :
$$\mathcal{AC}'' \subseteq (\mathcal{AC}\ \Lambda\ \mathcal{AC}') \text{ iff } \mathcal{AC}'' \subseteq (\mathcal{AC} \mid_V\ \Lambda\ \mathcal{AC}').$$

This proposition expresses that abstract projection does not cause loss of information. In particular, the abstract projection on V cannot invalidate the inferred freeness of variables in V.

Proposition 4.5 (Condition for constraint reordering)
Given a set of constraints $\mathcal{C}$ preceding a procedure call b, and an abstract constraint system $\mathcal{AC}$ approximating the state before executing $\{\mathcal{C}\}, b$. If $(\mathcal{AC}\ \Lambda\ \alpha(\mathcal{C})) \mid_{vars(b)} = \mathcal{AC} \mid_{vars(b)}$ then $\mathcal{C}$ can be moved after the call b.

5 A complete and practical analysis

The previous sections focus on the core of our analysis: the abstraction of constraint systems involving unifications and linear equations and the specification of the abstract conjunction and projection operations. This section describes the treatment of more general constraints and the other abstract operations, needed to obtain a complete analysis. A prototype implementation has been developed within the framework of Bruynooghe [1]. Both the precision as well as the efficiency of the analysis can be improved by incorporating results from other analyses [15, 10].

More general constraints

Linear disequations and inequalities of the form $a_1X_1 + \ldots + a_nX_n\ Op\ b$ with $Op \in \{>, \geq, \neq\}$ can be abstracted in the same way as linear equations. The worst case assumption for a solvable system is that the left hand side is constrained to a single value n (with $n >, \geq$ or $\neq b$).

Non-linear numerical constraints are more difficult to handle. Most of the current CLP systems delay such constraints until they become linear. A safe but imprecise way to abstract a non-linear constraint c is to take the powerset of $vars(c)$ as the abstraction (hence considering every variable in $vars(c)$ as possibly non-free). Other more precise ways to handle non-linear constraints are still under investigation. The treatment of non-linear constraints can certainly be improved by combining our abstraction with definiteness information as explained below.

Other abstract operations

The *procedure-entry* and *procedure-exit* operations of [1] are defined in terms of the previous operations. Due to lack of space, they are not discussed here. One subtle point regarding procedure-exit is worth mentioning. The result of analysing a clause body can be partitioned into *old* information that was passed on at procedure entry and *new* information gathered during local analysis of the clause body. Upon procedure exit, it is essential not to compose the old information with the information in the caller's environment, as this would destroy the freeness of the involved variables. In fact, as our abstraction discards the coefficients of numerical constraints, it is disastrous to inadvertently add a numerical constraint twice to the abstract constraint store. Consider a constraint $a_1X_1 + a_2X_2 + \ldots + a_nX_n = a_{n+1}$ which is abstracted as $\{X_1, \ldots, X_n\}$. Hence, the second time this constraint is added we effectively consider its linear combination with an equation represented by $\{X_1, \ldots, X_n\}$, i.e. an equation of the form $b_1X_1 + b_2X_2 + \ldots + b_nX_n = b_{n+1}$ where the b_i are arbitrary coefficients. With an adequate choice of the b_i, a linear combination can be obtained such that each X_i becomes possibly non-free. Making the distinction between new and old information in the

analysis of logic programs is applied previously by Plaisted [25] and also by Mulkers [22, 23].

An implementation

Our analysis has been implemented within the abstract interpretation framework of Bruynooghe [1]. As an example, we show the results obtained for the sumlist/2 program with an initial call pattern $sumlist(a, f)$ indicating that the first argument is *any* term and the second argument is a *free* variable. The abstract constraint system is written out at each program point.

Example 5.1

sumlist(L, S) :–	% $\{\ \{L\}\ \}$
$\{L = [\,],$	% $\{\ \{L\}\ \}$
$S = 0\}.$	% $\{\ \{L\}, \{S\}, \{L, S\}\ \}$
sumlist(L, S) :-	% $\{\ \{L\}\ \}$
$\{L = [X \mid T],$	% $\{\ \{L\}, \{L, X\}, \{L, T\}, \{L, X, T\}, \{X\}, \{T\}, \{X, T\}\ \}$
$S = S' + X\},$	% $\{\ \{L\}, \{L, X\}, \{L, T\}, \{L, X, T\}, \{X\}, \{T\}, \{X, T\},$
	% $\{S, S', X\}, \{S, S', X, L\}, \{S, S', L\}, \{S, S', X, L, T\},$
	% $\{S, S', L, T\}, \{S, S'\}, \{S, S', X, T\}, \{S, S', T\}\ \}$
sumlist(T,S').	% $\wp(\{L, S, X, T, S'\}) \setminus \{\phi\}$

The analysis indicates that at the end of the first clause, both L and S are possibly non-free. In the second clause, S and S' are free before the recursive call. Moreover, $\{S = S' + X\}$ does not modify the projection of the system on T and S', so it can safely be shifted after the recursive call.

Refinements and future work

Abstracting linear inequalities and disequations as defined above has a rather disadvantageous effect on the precision : a constraint $N \geq 0$ is abstracted as $\{N\}$ and this destroys the freeness of N. The work in [15] describes how to recognise many such constraints as future redundant, implying that they do not need to be abstracted.

To obtain the power of a complete mode analysis system, our analysis should be combined with definiteness information [20, 10] (usually called groundness information in the context of logic programs). Definiteness information can improve the abstraction of non-linear constraints : it allows to detect whether a syntactically non-linear constraint has become linear (in which case it can be abstracted accordingly). Definiteness information is also needed - in addition to freeness information - to specialise constraints (e.g. to turn constraints into tests or assignments). Recall that our analysis also contributes in another way to constraint specialisation: the information in the abstract constraint systems allows to delay the addition of constraints to the store - in other words, to shift constraints - which leads to more opportunities for subsequent specialisation as will be shown below. Finally,

if definiteness information is available, our abstract constraint systems can be compressed considerably: it then suffices to keep track of dependencies between variables having mode free or any, while variables being definitely constrained to a unique value can be separated out. This is needed to obtain an efficient and practical analyser.

Example 5.2
Reconsider the sumlist example. Combining the results of example 5.1 with definiteness information yields the following results for the call pattern sumlist(d,f) (the modes[4] *and compressed abstract constraint systems are given at each program point):*

```
sumlist(L, S):-          % L:d, S:f; AC = φ
    { L = [],            % L:d, S:f; AC = φ
    S = 0 }.             % L:d, S:d; AC = φ
sumlist(L, S):-          % L:d, S:f, X:f, T:f, S':f; AC = φ
    { L = [X | T],       % L:d, S:f, X:d, T:d, S':f; AC = φ
    S = S' + X }         % L:d, S:f, X:d, T:d, S':f; AC = { {S, S'} }
    sumlist(T,S'),       % L:d, S:d, X:d, T:d, S':d; AC = φ
```

This information allows to obtain the following optimised program:

```
sumlist(L, S):-
    L = [],              % test
    S := 0.              % assignment
sumlist(L, S):-
    L = [X | T],         % argument selection
    sumlist(T,S'),
    S := S' + X.         % shifted constraint, specialised into assignment
```

The results in [15] already indicate that specialising constraints into tests or assignments yield significant speed-ups. Our experiments with the CLP($\mathcal{R}$) system (version 1.2) have shown that shifting constraints (followed by constraint specialisation) allows to improve the efficiency even more. For the sumlist example, the optimised program yields a speed-up of 28.6% with respect to the general program of example 5.1. If a program gives rise to independent inequalities (i.e. inequalities over distinct variables), then the improvement can go up to 61.5% (as the expensive simplex algorithm is avoided). For a general system of dependent equations and inequalities the optimisation really pays off (speed-up up to 98%; it then strongly depends on the number of variables involved in the constraint system).

6 Discussion

The abstraction introduced in this paper provides information that enables moving constraints in clause bodies, similar to the reordering optimisation

[4] mode 'f' abstracts a free variable, 'd' abstracts a variable definitely constrained to a unique value and 'a' abstracts any term

suggested in [21]. In contrast, our abstract operations are computationally less expensive than the use of the convex hull and Fourier approximations proposed in [21]. Moreover, the condition for moving constraints does not involve a satisfiability check and appears to be substantially cheaper. In addition, our approach supports the abstraction of mixed systems of unification and numerical constraints. One drawback is that the abstractions tend to become large under the abstract operations. However, combining the analysis with definiteness information substantially reduces their size. There are two ways to introduce definiteness information. First of all, the technique proposed by Cousot [7] and discussed in [3] in the context of LP allows to combine our domain with the one in [10] without having to redefine the operations of the two domains. An alternative approach to derive definiteness information without keeping track of *definite* dependencies between variables as in [10] is to apply the "reexecution" strategy of [1, 17]. In pure logic programs, a procedure call can be reexecuted without affecting the current state. Experiments in [17] show that reexecution over a simple abstract domain often gives as accurate results as an analysis using a more complex domain. Consider a linear equation $aX + bY = c$. At a later point, when X becomes ground, reexecuting the concrete linear equation allows to infer that Y becomes *definitely* ground. Note that in this case reexecution is used only to propagate definiteness and will not be applied on the possible dependency component of our domain (where it could destroy freeness).

Finally, besides integrating definiteness information, a further improvement of the precision of the analysis can be obtained by delaying the abstraction of numerical constraints and to use their concrete form as long as possible. In fact, they only need to be abstracted when the concrete coefficients are unknown, and in treatment of recursion when widening [8] has to be applied to avoid (infinite) ascending chains. The reexecution method mentioned above is also considered in this respect.

Acknowledgements

This research was supported in part by Diensten voor de Programmatie van het Wetenschapsbeleid (project RFO-AI-02 : "Logic as a basis for artificial intelligence : control and efficiency of deductive inference - parallelism") and ESPRIT project 5246 PRINCE; the third author is supported by the Belgian National Fund for Scientific Research.

References

[1] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, Feb. 1991.

[2] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[3] M. Codish, A. Mulkers, M. Bruynooghe, M. G. de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, June 1993. (to appear).

[4] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 30(7):52–68, 1990.

[5] A. Colmerauer. An introduction to PROLOGIII. *Communications of the ACM*, 30(7):69–96, 1990.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, 1977.

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings 6th ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.

[8] P. Cousot and R. Cousot. Comparing the Galois connection with widening/narrowing approaches to abstract interpretation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, Leuven, Belgium, August 1992.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[10] M. G. de la Banda and M. Hermenegildo. A practical approach to the global analysis of constraint logic programs. Technical Report UPM/DIA/CLIP/PRINCE-1/92.1, UPM, Madrid, April 1992.

[11] R. Giacobazzi, S.K.Debray, and G. Levi. A generalized semantics for constraint logic programs. In *Proceedings of the international conference on fifth generation computer systems*, pages 581–591, Tokyo, Japan, June 1992.

[12] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the Fourteenth ACM symposium of the principles of programming languages*, pages 111–119, Munich, 1987.

[13] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) language and system. IBM Research Report RC 16292(#72336), 1990.

[14] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. An abstract machine for CLP(R). In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 128–139, San Francisco, June 1992.

[15] N. Jørgensen, K. Marriott, and S. Michaylov. Some global compile-time optimizations for CLP(R). In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 420–434, San Diego, 1991.

[16] J.-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.

[17] B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of prolog. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 750–764. MIT Press, November 1992.

[18] J. W. Lloyd. *Foundations of Logic Programming.* Springer Series : Symbolic Computation - Artificial Intelligence. Springer-Verlag, second, extended edition, 1987.

[19] A. Mariën, G. Janssens, A. Mulkers, and M. Bruynooghe. The impact of abstract interpretation: An experiment in code generation. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 33–47, Lisbon, 1989. MIT Press, Cambridge.

[20] K. Marriott and H. Søndergaard. Analysis of constraint logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 531–547, Austin, 1990. MIT Press, Cambridge.

[21] K. Marriott and P. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, Removal and Reordering. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 334–344, Charleston, South Carolina, January 1993.

[22] A. Mulkers. *Deriving Live Data Structures in Logic Programs by Means of Abstract Interpretation.* Ph.D. thesis, Department of Computer Science, Katholieke Universiteit Leuven, Dec. 1991.

[23] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of shared data structures for compile-time garbage collection in logic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 747–762, Jerusalem, 1990. MIT Press, Cambridge.

[24] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[25] D. A. Plaisted. The occur-check problem in Prolog. *J. New Generation Computing*, 2(4):309–322, 1984. Also in: *Proceedings of the International Symposium on Logic Programming*, pages 272–280, Atlantic City, 1984. IEEE Computer Society Press.

[26] A. Taylor. LIPS on a MIPS, results from a Prolog compiler for a RISC. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 174–185, Jerusalem, 1990. MIT Press, Cambridge.

[27] P. Van Roy and A. M. Despain. High-Performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, pages 54–67, 1992.