A Practical Approach to the Global Analysis of CLP Programs

M. García de la Banda
M. Hermenegildo
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - Spain
{maria,herme}@fi.upm.es

Abstract

This paper presents and illustrates with an example a practical approach to the dataflow analysis of programs written in constraint logic programming (CLP) languages using abstract interpretation. It is first argued that, from the framework point of view, it suffices to propose relatively simple extensions of traditional analysis methods which have already been proved useful and practical and for which efficient fixpoint algorithms have been developed. This is shown by proposing a simple but quite general extension of Bruynooghe's traditional framework to the analysis of CLP programs. In this extension constraints are viewed not as "suspended goals" but rather as new information in the store, following the traditional view of CLP. Using this approach, and as an example of its use, a complete, constraint system independent, abstract analysis is presented for approximating definiteness information. The analysis is in fact of quite general applicability. It has been implemented and used in the analysis of CLP(R) and Prolog-III applications. Results from the implementation of this analysis are also presented.

1 Introduction

In Constraint Logic Programming (CLP) languages programs can perform computations over both symbolic and non-symbolic domains and unification is replaced by the concept of constraint solving [11]. While this greatly enhances expressive power, constraint solving can often be much more expensive than unification and result in low run-time performance. In addition, current CLP systems are often also significantly slower than Prolog systems when running equivalent (i.e. "Prolog") programs. Such performance limitations, combined with the increasing acceptance of these languages, have motivated a growing interest in dataflow analysis based optimization techniques for CLP languages, and in particular in the application of abstract interpretation [5].

Much work has been done using the abstract interpretation technique in the context of logic programs (e.g. [16, 7, 1, 14, 6]). A number of practical systems have been built, some of which have shown great usefulness and practicality [18, 19, 17, 6, 3]. It appears that the abstract interpretation technique should also be useful in the context of CLP.

A few general frameworks have already been defined for this purpose [15, 4, 2]. However, one common characteristic of these frameworks is that they depart from the approaches that have been so far quite successful in the analysis of traditional logic programing (LP) languages. It is the point of this paper to show how some of the techniques which have been used to great success and practicality in LP and for which efficient fixpoint algorithms have already been developed can relatively easily be extended to the analysis of CLP programs.

The point above is illustrated by proposing a simple but quite general and powerful extension of Bruynooghe's traditional framework in order to make it applicable to the analysis of CLP programs. In this extension constraints are viewed not as "suspended goals" (unless of course they actually are implemented through suspension in the concrete semantics, as may be the case for example for non-linear constraints) but rather as new information in the store, following the view of the traditional CLP framework. We give correctness conditions for the resulting generalized framework. We also argue that given such a framework, the effort should then concentrate on the development of accurate abstract domains and abstract conjunction functions. We then show how one of the key issues in achieving this is the accurate abstraction of the entailment relation. We also relate this point to traditional issues in the Herbrand domain. As an example of the use of this approach, a complete, constraint system independent abstract analysis is presented for approximating definiteness information. The analysis is of quite general applicability since it uses in its implementation only constraints over the Herbrand domain. We also present some encouraging results from the implementation of this analysis.

2 Preliminaries

Let us present some basic concepts of constraint logic programming and the notation which will be used throughout the paper. We follow mainly [11].

Let F be a set of function symbols, V a set of variables, $\Pi = \Pi_C \cup \Pi_P$ a set of predicate symbols, where Π_C are the constraint predicates including the symbol "=" and $\Pi_C \cap \Pi_P = \emptyset$. Let (F)-terms and $(F \cup V)$-terms be the set of ground and possibly non ground *terms* respectively, $Atomic = (\Pi_C, F \cup V)$-atoms be the set of *primitive constraints*, and $Atom = (\Pi_P, F \cup V)$-atoms be the set of atoms. A *constraint* is a (possibly empty) set of primitive constraints that will be interpreted as the conjunction of its elements. A *literal* is an atom or a primitive constraint.

Constraints are pre-ordered by logical implication, that is $\pi \leq \pi'$ iff $\pi \Rightarrow \pi'$. For simplicity, and similarly to the consideration of only idempotent substitutions, we will just consider constraints that are closed under entailment. We let $\exists_W \pi$ be a non-deterministic function which returns a constraint logically equivalent to $\exists V_1 \exists V_2 \cdots V_n \pi$ where variable set $W = \{V_1, \ldots, V_n\}$. We let $\overline{\exists}_W \pi$ be constraint π restricted to the variables W. That is $\overline{\exists}_W \pi$ is $\exists_{vars(\pi) \setminus W} \pi$ where function *vars* takes a syntactic object and returns the set of (free) variables occurring in it. Note that $\overline{\exists}_W \theta \Leftrightarrow \theta$ in the traditional Logic Programming framework would be equivalent to saying that $domain(\theta) = W$. In the spirit of this concept, in the following we will say that $domain(\pi) = W$ iff $\overline{\exists}_W \pi \Leftrightarrow \pi$.

A Constraint Logic Program is a finite set of clauses of the form $Head \leftarrow$

Body, where *Head* is an atom and *Body* is a sequence of the form $Q_1, \cdots, Q_n$, where each Q_i is a literal. A *goal* is a (possibly empty) sequence of literals.

A state $\langle G, \pi \rangle$ consists of the current sequence of goals G and the current constraint π. We will say that π is the call of G. A (generalized) derivation step of state $s = \langle L : G, \pi \rangle$ for program P returns a state s' such that:

1. if $L \in Atomic$ and $\exists (L \wedge \pi)$, $s' = \langle G, \{L\} \uplus \pi \} \rangle$
2. if $L \in Atom$, and exists a clause $C:\ H \leftarrow B$ s.t. $vars(C) \cap vars(s) = \emptyset, \exists \pi \wedge L = H$ then $s' = \langle B : G, \pi' \rangle$ where $\pi' = \pi \wedge L = H$

The derivation of a state s for a program P is a finite or infinite sequence of states $s_0 \rightarrow s_1 \rightarrow \cdots$ returned by derivation steps, in which $s_0 = s$. It is *successful* when the last state has an empty sequence of atoms. A constraint π is a *partial answer* to state s if there is a derivation from s to a state with constraint π. An *answer* to state s is a partial answer corresponding to a successful derivation.

3 Towards a CLP Analysis Framework

There has been considerable interest in developing new abstract interpretation frameworks for CLP languages. To these authors' knowledge, at least three frameworks have been proposed previously or simultaneously with our work.[1] Marriott and Sondergaard [15] present a general and elegant, semantics based framework. It is based on a definition-independent meta-language which can express the semantics of a wide variety of programming languages including CLP languages. However, from a practical point of view, this framework does not provide much simplification to the developer of the abstract interpretation system, in the sense that many issues are left open.

In fact, one of the advantages of the most popular methods used in the analysis of conventional LP systems (for example Bruynooghe's method [1] and the optimizations proposed for it [17]) is that they are "generic," in the sense that they specify much of what is needed leaving only the definition of the domain, domain dependent functions, and assurance of correctness criteria to be provided by the implementor. It is our intention to develop a framework for CLP program analysis at this level of specification.

Codognet and Filé [4] also present a quite general framework for the description of both CLP languages and their static analyses and an implementation approach. Although more concrete, this proposal is still more abstract than the level pointed out above as our objective. On the other hand this paper introduces the quite interesting idea of implementing the abstract functions actually using constraint solvers, to which we will return later.

Finally, Bruynooghe and Janssens [2] present a specialized framework (which has been developed in parallel with the proposal presented in this paper) which is based on the idea of adding complexity to the framework with the potential benefit of decreased complexity in the abstract domain. This is done by incorporating a local form of "suspension" so that some

[1]This ideas illustrated in this paper were first presented at the ICLP'91 Workshop on Constraint Logic Programming.

goals can be reconsidered if later execution in a different environment can provide further information. This extension is based on a particular view of the execution of a CLP program in which constraints are considered as goals which can suspend depending on the state of its arguments and on the particular constraint system.

The view of constraints as suspended goals is certainly interesting and worth pursuing. However, we feel that, understandably, this makes it more difficult to make the framework fully general and we prefer to take the more traditional notion presented in the CLP scheme in which constraints take the place of substitutions and goals always either succeed or fail, in the former case possibly *placing* new constraints.[2]

One of the main points of this paper is to show that if the above view is taken then standard abstract interpretation frameworks for logic programs are essentially still useful for the analysis of constraint logic programs, provided the parts that relate to the abstraction of the Herbrand domain and unification functions are suitably generalized. This is based on the fact that in this traditional view the role of goals and their control is basically identical to those in traditional LP systems, the differences being essentially limited to replacing the notions of Herbrand domain, unification, and substitutions by those of constraint system, conjunction, and constraints.

In particular, we argue that the traditional framework of Bruynooghe and its extensions can be used for analyzing constraint logic programs by using the notions of abstract constraint and abstract conjunction and reformulating the safety conditions, but keeping the construction of the AND-OR tree, the implementation and optimizations of the fixpoint algorithm, the notions of projection and extension, etc. This has the advantage that the relatively large number of implementations based on this scheme or derivations thereof can be applied to CLP systems provided the safety conditions and other related requirements proposed herein are observed. In section 4 we propose exactly such an extension. We now present some of the motivations behind the approach taken.

3.1 Accurate Abstraction of Entailment as a Key Issue

Assume we want to analyze a language using a particular constraint system in order to obtain information regarding a given property. Usually, we will define (1) an abstract domain which represents this property, (2) an abstraction function which maps a constraint into the abstract domain, and (3) an abstract conjunction function which approximates the concrete solver algorithm, i.e. a function which takes as arguments an abstract constraint store and an abstract constraint and obtains the resulting abstract constraint store.

Typically, abstracting data implies losing information. This is not important if the information lost is not relevant to the particular property of interest. In fact, losing non relevant information is desirable in order to reduce the amount of information that has to be handled by the analyser.

[2]In fact, actual suspension, as is often used in the solving of non-linear arithmetic constraints or in programs with explicit coroutining can also be modeled in this way. However, we propose treating actual suspension directly using techniques such as those proposed for analyzing programs with delay declarations.

However, determining which information is relevant to the property is being abstracted is not always easy. As an example, assume we are interested in knowing if a program variable is definitely free. One could think that this property can be accurately abstracted by defining (1) above as the set of variables which are definitely free in a constraint. This is true w.r.t. (1) and (2), i.e. for each constraint we will obtain the most accurate representation w.r.t. the property desired. However, given this abstraction, in order to be safe we will necessarily often lose almost all the information when applying the abstract conjunction function. The reason is that, no matter how this abstract function is defined, we will not be able to approximate the way in which the underlying constraint solver propagates non freeness and therefore we will have to assume that all variables would become non free. Thus, it is clear that in this abstraction we have lost *relevant* information: the information which allows us to accurately approximate non freeness propagation.

This problem occurred with early analyzers for LP which in fact inferred less accurate (or in some cases incorrect) information due to the lack of propagation (referred to often also as tracking "aliasing"). It is now clear that most properties such as groundness, freeness, etc. in the Herbrand constraint system can be propagated quite accurately via a form of "sharing" and thus, abstracting sharing provides a more accurate analysis. However, not all properties need the same abstraction of sharing. Groundness, for example, only needs covering, as exemplified by *Prop* [14]. On the other hand, freeness needs a more general abstraction such as "possible sharing."

The problem then is to define which is the relevant information for each property. We argue that in CLP terms this relevant information is nothing more than the information needed to abstract the entailment relation associated with this property. Note that this does not imply explicitly abstracting the information provided by all possible entailed constraints, but rather that which plays a role in preserving the characteristics of the concrete entailment w.r.t. the target property. Once the relevant information is identified, the level of accuracy in its abstraction, and therefore the level of accuracy of the associated abstract conjunction function, can be chosen as determined by the desired trade-off between efficiency and accuracy. Returning to the problems with early analyzers for LP and "aliasing," note that after analyzing the goals $X = Y$, $Y = Z$, and $Z = a$ if X is inferred (incorrectly) to be a free variable or (inaccurately) to be $\top$, the problem can now be seen as related to not taking into account the entailed relation $X = Z$ which is relevant to the propagation of freeness information.

3.2 Developing Analyses for Practical Languages

Although using the ideas sketched above, and as we will show, extending the abstract frameworks can be considered a relatively simple task, developing abstract domains and the corresponding abstract functions capable of accurately and correctly abstracting properties of the constraint systems can sometimes be quite involved.

Satisfying the correctness conditions in traditional LP languages is "reasonably" simple since the Herbrand domain with the equality constraint is the only constraint system and the unification algorithm is well known. Therefore, the condition only implies "correctly propagating via sharing" the desired properties, knowing that the accuracy of the sharing abstraction determines in some sense the accuracy of the inferred information. However, in CLP languages which include other constraint systems additional complications arise.

The first such complication is related to the intrinsic complexity associated with most of the constraint solver algorithms which implies that an accurate analysis can sometimes be so complex so as to result intractable. An interesting issue from a practical point of view is the vehicle to be used for implementing the abstract conjunction. As mentioned before, Codognet and Filé propose the direct use of CLP solvers in specifying the abstract solving algorithms. The use of the constraint solving capabilities of the implementation language is a very elegant solution and has the advantage that the abstract algorithm can be specified in a declarative way. On the other hand, and from a practical point of view, as is the flavour of this paper, one favourable aspect of formulating analyses so that they can be executed using only equalities over the Herbrand domain is generality, since it will be quite simple to implement them on a large number of CLP systems (and traditional logic programming systems!), given that in general all CLP systems include the Herbrand domain and a unification algorithm.

A more subtle complication in developing analyses for CLP comes from the fact that most of CLP languages are defined over several constraint systems, and in most cases the theoretical separation among the objects (functors, constraint predicates, domain variables, etc.) of each constraint system is not maintained. For example the constraint $X = Y$, where X and Y are variables, can belong to almost any constraint system. Also, the numerical constraint $X = Y + Z$ can affect a variable W in the Herbrand domain if it is defined as, for example, $W = f(X, Y)$. Therefore, it is not only necessary to abstract the constraint solver algorithm for each constraint system but also the effects that the conjunction of a particular constraint can produce with respect to any of the other constraint systems in the language. Another example, in addition to the one given in this paper, of how this can be done can be found in [8], which is based in part on the ideas developed in this paper.

The considerations given above suggest the development of a hierarchy of domains and analyses where there is a top-level domain applicable to all constraint systems and some lower level domains which are constraint system specific. The top level domain is then used for performing the transfer of information among the lower level domains that is necessary in order to preserve correctness and achieve reasonable efficiency.

4 Extension of the framework

In this section we formalize the extension of the framework presented in [1] and provide safety conditions to be met by the user-defined functions. We will mainly follow the notation and scheme of [12] in which a summary of the correctness conditions required in this framework is given.

4.1 The derivation scheme

The derivation schema mentioned in the preliminaries is changed in the framework of [1] in that it only considers states $s = \langle g_i : \cdots : g_n, \pi \rangle$ in which the sequence of literals $g_i : \cdots : g_n$ is either the tail of a body of a clause or the initial query, and $domain(\pi)$ is a subset of the variables occurring in respectively the clause or the query.

Consider a derivation of the state $s = \langle A_1 : \cdots : A_n, \epsilon \rangle$, ϵ being the empty substitution, which has reached the state $s_i = \langle A_i : \cdots : A_n, \pi_i \rangle$, where $domain(\pi_i) \subseteq vars(A_1 : \cdots : A_n)$. Let $C : H \leftarrow B_1, \cdots, B_m$ be a clause s.t. $vars(C) \cap vars(s) = \emptyset$ and $\pi_i \wedge A_i = H$ is satisfiable. The new schema will proceed in the following steps. If $A_i \in Atoms$:

- $project(A_i, \pi_i)$: obtains $\pi_{proj} = \overline{\exists}_{vars(A_i)} \pi_i$.
- $procedure\_entry(C, A_i, \pi_{proj})$: obtain $s^1 = \langle B_1 : \cdots : B_m, \pi_{in} \rangle$ where π_{in} is $\overline{\exists}_{vars(C)}(\pi_{proj} \wedge (A_i = H))$.
- $procedure\_exit(C, A_i, \pi_i)$: assume that after some subderivations, we obtain the answer π_{answ}, $domain(\pi_{answ}) \subseteq vars(C)$. Then, we obtain $\pi_{out} = \overline{\exists}_{vars(H)} \pi_{answ}$ and $\pi_{exit} = \overline{\exists}_{vars(A_i)}(\pi_{out} \wedge \pi_i \wedge (A_i = H))$.
- $procedure\_extend(s, \pi_i, \pi_{exit})$: obtain $s_{i+1} = \langle A_{1+1} : \cdots : A_n, \pi_{i+1} \rangle$, where $\pi_{i+1} = \pi_i \wedge \pi_{exit}$

If $A_i \in Atomic$, we only need the functions, $project(A_i, \pi_i)$ (as above), $procedure\_constraint(A_i, \pi_{proj}) = \pi_{exit} = A_i \wedge \pi_{proj}$ and $procedure\_extend(s, \pi_i, \pi_{exit})$ (as above).

Assume that $\pi_i = \overline{\exists}_{vars(A_1 : \cdots : A_n)} \psi_i$, ψ_i being the constraint obtained with the original scheme. Also assume that $\psi_{i+1} = \psi_i \wedge \pi'$. We have to prove that π_{i+1} is a solvable restriction of $\overline{\exists}_{vars(A_1 : \cdots : A_n)} \psi_{i+1}$. If $A_i \in Atom$:

$$\begin{aligned}
\pi_{i+1} &= \pi_i \wedge \pi_{exit} \\
&= \pi_i \wedge \overline{\exists}_{vars(A_i)}(\pi_{out} \wedge \pi_i \wedge (A_i = H)). \\
&= \pi_i \wedge \overline{\exists}_{vars(A_i)}(\overline{\exists}_{vars(H)} \pi_{answ} \wedge \pi_i \wedge (A_i = H)) \\
&= \pi_i \wedge \overline{\exists}_{vars(A_i)}(\pi_{answ} \wedge \pi_i \wedge (A_i = H)) \\
&= \overline{\exists}_{vars(A_1 : \cdots : A_n)}(\pi_{answ} \wedge \pi_i \wedge (A_i = H)) \\
&= \overline{\exists}_{vars(A_1 : \cdots : A_n)}(\psi_i \wedge \pi_{answ} \wedge (A_i = H)) \\
&= \overline{\exists}_{vars(A_1 : \cdots : A_n)} \psi_{i+1}
\end{aligned}$$

A similar (simplified) proof can be derived if $A_i \in Atomic$.

4.2 The Abstract Domain

Let δ be an abstract constraint defined over the variables D of a clause/query. Let Abs_D be the set of abstract constraints which are defined over D. The abstract interpretation framework requires Abs_D to have:

1. a preorder $\sqsubseteq$ satisfying $\forall \delta_1, \delta_2 \in Abs_D . \delta_1 \sqsubseteq \delta_2 \Rightarrow (\gamma(\delta_1) \Rightarrow \gamma(\delta_2))$

2. an upper bound upp satisfying $\forall \delta_1, \delta_2 \in Abs_D \Rightarrow \exists upp(\delta_1, \delta_2) \in Abs_D$ and $\delta_1 \sqsubseteq upp(\delta_1, \delta_2).\delta_2 \sqsubseteq upp(\delta_1, \delta_2)$

3. a maximal element δ_{max} s.t. $\forall \delta \in Abs_D.\delta \sqsubseteq \delta_{max}$

4. a minimal element $\perp$ s.t. $\gamma(\perp) = \emptyset.\forall \delta \in Abs_D.\perp \sqsubseteq \delta$

5. $F_D \subseteq Abs_D$ s.t. F_D has no infinite ascending chain for $\sqsubseteq$, $\delta_{max}, \perp \in F_D$

6. an operator $R : Abs_D \longrightarrow F_D$ satisfying $\delta \sqsubseteq R(\delta)$

Note that all conditions are identical to those given in [12] except the first one in which the general pre-order for constraints is used instead of the particular pre-order for substitutions.

4.3 The Abstract operations

As in [12] we will assume in the following that any literal has the form $A(X_1, \cdots, X_s)$ with X_i being distinct variables.[3] Let $A_1, A_2, \cdots$ denote literals and π, δ (with or without suffix) denote respectively concrete and abstract constraints. Then the sufficient conditions for correctness of each step in the scheme are:

1. $abstract\_project(A_i, \delta_i) = \delta_{proj}$:
   $\exists \pi \in \gamma(\delta_i) \Rightarrow \overline{\exists}_{vars(A_i)} \pi \in \gamma(\delta_{proj})$

2. $abstract\_procedure\_entry(C, A_i, \delta_{proj}) = \langle B_1 : \cdots : B_m, \delta_{in} \rangle$:
   $\exists \pi_{proj} \in \gamma(\delta_{proj}) \Rightarrow \overline{\exists}_{vars(C)} (\pi_{proj} \wedge (A_i = H)) \in \gamma(\delta_{in})$

3. $abstract\_procedure\_exit(C, A_i, \delta_{proj}) = \delta_{exit}$:
   $\exists \pi_{out} \in \gamma(\delta_{out}).\exists \pi_i \in \gamma(\delta_i) \Rightarrow \overline{\exists}_{vars(A_i)} (\pi_{out} \wedge \pi_i \wedge (A = H)) \in \gamma(\delta_{exit})$
   $abstract\_procedure\_extend(s, \delta_i, \delta_{exit}) = s_{i+1} = \langle A_{1+1} : \cdots : A_n, \delta_{i+1} \rangle$
   $\exists \pi_i \in \gamma(\delta_i) \Rightarrow \exists \pi_1 \in \gamma(\delta_{exit}).(\pi_i \wedge \pi_1) \in \gamma(\delta_{i+1})$

4. $abstract\_procedure\_constraint(A_i, \delta_{proj}) = \delta_{exit}$:
   $\exists \pi_{proj} \in \gamma(\delta_{proj}) \Rightarrow (\pi_{proj} \wedge A_i) \in \gamma(\delta_{exit})$

Note that if those functions are defined in terms of the abstract conjunction and the abstract projection functions, then all conditions are satisfied if the abstract projection satisfies condition 1 and the abstract conjunction function Λ satisfies that $\forall \delta_1, \delta_2 \in A_D, \exists \pi_1 \in \gamma(\delta_1), \exists \pi_2 \in \gamma(\delta_2) \Rightarrow (\pi_1 \wedge \pi_2) \in \gamma(\Delta(\delta_1, \delta_2))$.

Comparing this extended framework with the original, it is clear that on one hand only one more function (*abstract_procedure_constraint*) has been defined and, on the other hand, the conditions required by the abstract functions are natural generalizations from LP into CLP, which actually introduces a certain notational simplification. The former is not surprising

[3]In fact, in this respect the traditional CLP scheme provides a slight technical simplification w.r.t. LP in that programs are generally assumed to be in normal form by the scheme, even if actual languages do support as syntactic sugar the introduction of constraints in atoms.

since the operational behaviour of a CLP program is almost equivalent to that of a traditional LP program, except when a constraint literal is considered. The latter is not surprising either since the only change is that, instead of representing a particular kind of constraint (substitutions), we are representing constraints over different constraint systems.

5 An Example: Inference of Definiteness Information

In order to illustrate the ideas presented in previous sections we present a simple abstract analysis for inference of definiteness information in CLP programs. The abstraction is based on a high-level description of uniquely constraining patterns which is then easy to obtain for each particular type of constraint in an actual system. This domain can be seen as an encoding and implementation of the *Prop* domain defined in [14] for traditional logic programming languages, but without disjunction. The collapsing of disjunctive information has been done in order to reduce the size of abstract constraints.[4]

We define the abstract domain and abstract functions required for the framework developed above. The strong relation with the *Prop* domain, which has been proved correct allows us to provide a clear (and brief) intuition of the scheme of each proof, by showing how our domain correctly abstracts *Prop*.

Definiteness information is abstracted by keeping for each program variable X those sets of program variables which, if they become uniquely constrained, constrain X to have a unique value. Then, an abstract constraint will be an element of $Domain = \wp(\wp(Pvar \times \{d, \wp(\wp(Pvar)), \top\}))$, i.e. a set of couples of the form (X, SS) such that $SS \in \{d, \wp(\wp(Pvar)), \top\}$. Therefore, translating an abstract constraint into the *Prop* domain is as simple as obtaining the conjunction of the constraints represented by each element (X, SS). These constraints can be obtained in the following way: if $SS \in \wp(\wp(Pvar))$, then $\forall S \in SS, A_i \in S$ we have the constraint $A_1 \wedge \cdots \wedge A_n \rightarrow X$, if $SS = d$ then we will have $X = 1$, and if $SS = \top$ then we will have *true*.

Let us first define some simple functions which will be used to eliminate unnecessary complexity from the domain dependent functions. We will denote by $\pi, \pi_1, \cdots$ constraints, and by $def(\pi) \subseteq Var$ the set of uniquely constrained variables in π.

The function $min(SS)$ takes as argument a set of sets of variables $SS \in \wp(\wp(Var))$, and returns the set of sets of variables which results from eliminating all supersets from SS. Formally,

$$min(SS) = \{S \in SS | \neg \exists S' \in SS, s.t.\ S' \subset S\}$$

The function $constrain(\pi, X)$ takes as arguments a constraint π and a free variable $X \in vars(\pi)$ and returns the minimized set of sets of variables

[4]Hanus has recently and independently proposed an analysis of definiteness which is quite closely related to the instantiation that we propose for definiteness analysis of our extension of Bruynooghe's framework [10].

which uniquely constraining them, uniquely constrains X. Formally, let $X \in Var, X \in vars(\pi)$,

$$constrain(\pi, X) = min(SS)$$

where $SS = \{S \in \{\wp(vars(\pi)) \setminus \{X\}\} |$ uniquely defining all $Y \in S$, uniquely defines $X\}$

Example 5.1

$$\begin{array}{lclcl}
constrain(X = f(Y,Z), Y) & = & min(\{\{X\},\{X,Z\}\}) & = & \{\{X\}\} \\
constrain(X = Y + Z, Y) & = & min(\{\{X,Z\}\}) & = & \{\{X,Z\}\} \\
constrain(X > Y + Z, Y) & = & min(\{\emptyset\}) & = & \{\emptyset\}
\end{array}$$

Intuitively, while the *constrain* function captures the definiteness information for the free variables in a constraint, the function $min(SS)$ simplifies the inferred information by eliminating redundant abstract constraints.

The function $restrict(Var, SS)$ takes as arguments a set of variables Var and a set of sets of variables SS and returns the set of sets of variables in SS which are subsets or equal to Var (this function will be used for abstracting the *procedure_project* function).

$$restrict(Var, SS) = \{S \in SS | S \subseteq Var\}$$

The function *prop_each* takes as arguments a couple $(X, SS1)$, where $X \in Pvar$ and $SS1 \in \{d, \wp(\wp(Pvar)), \top\}$, and an element δ of *Domain* and propagates the information in $(X, SS1)$ to δ in the following way:

$$prop\_each((X, SS1), \delta) = \{(X, SS1)\} \cup \{(Y, SS2) | \forall (Y, SS) \in \delta, Y \neq X\}$$

$$where \;\; SS2 = \begin{cases} if & X \notin vars(SS) \; or \; SS1 = \top & then \;\; SS \\ else\ if & SS1 = d, \exists S \in SS, S = \{X\} & then \;\; d \\ else\ if & SS1 = d & then \;\; Without \\ else & & min(Updated) \end{cases}$$

$Without = \{S' | \forall S \in SS, S' = S \setminus \{X\}\}$
$Updated = SS \cup \{\{S \setminus \{X\}\} \cup S' | S \in SS, X \in S, S' \in SS1, Y \notin S'\}$

Example 5.2

$$\begin{array}{ll}
prop\_each((X,d), \{(X,\top), (Y, \{\{X\}\})\}) & = \{(X,d), (Y,d)\} \\
prop\_each((Y, \{\{Z\}\}), \{(X, \{\{Y,Z\}\}), (Y,\top\}), (Z,\top)\}) & = \{(X, \{\{Y\},\{Z\}\}), \\
& \quad (Y, \{\{Z\}\}), (Z, \top\}
\end{array}$$

Intuitively, when the information for a program variable has changed, this function propagates this information to the rest of elements in the abstract constraint, possibly simplifying the abstract constraint.

5.1 Abstract Domain

An abstract constraint δ of the abstract domain *Def* is an element of $\wp(\wp(Pvar \times \{d, \wp(\wp(Pvar)), \top\}))$ satisfying: $\neg\exists(X, \{\emptyset\}) \in \delta$, $\exists(X, SS) \in \delta, SS \in \wp(\wp(Pvar)) \Rightarrow min(SS) = SS$, and $\exists(X, SS) \in \delta \Rightarrow prop\_each((X, SS), \delta) = \delta$.

Example 5.3

$\delta = \{(Z, \{\{X\}, \{X, Y\}\}), \cdots\} \notin Def$ since $min(\{\{X\}, \{X, Y\}\}) = \{\{X\}\}$
$\delta = \{(X, \{\{Y\}\}), (Y, d)\} \notin Def$ since $prop\_each((Y, d), \delta) \neq \delta$

Definition 1 (Abstraction of a constraint)

$$\Delta(\pi) = \{(X, SS) | \forall X \in vars(\pi)\}$$

$$where \;\; SS = \begin{cases} if & X \in def(\pi) & then \;\; d \\ elseif & constrain(\pi, X) = \emptyset & then \;\; \top \\ else & & constrain(\pi, X) \end{cases} \Box$$

Example 5.4

$$\begin{array}{lcl} \Delta(X = f(Y, Z)) & = & \{(X, \{\{Y, Z\}\}), (Y, \{\{X\}\}), (Z, \{\{X\}\})\} \\ \Delta(X :: N) & = & \{(X, \{\emptyset\}), (N, \{\{X\}\})\} \end{array}$$

where $X :: N$ constrains the length of the list X to be N.

Definition 2 (Partial order)

$\delta_1, \delta_2 \in Def, \delta_2 \sqsubseteq \delta_1$ *iff* $\forall (X, SS_1) \in \delta_1, \exists (X, SS_2) \in \delta_2$ *s.t.* $SS_1 = SS_2$ *or* $SS_2 = d$ *or* $SS_1 = \top$ *or* $SS_1, SS_2 \in \wp(\wp(Pvar)), \forall S1 \in SS_1, \exists S2 \in SS_2,$ *s.t.* $S2 \subseteq S1 \Box$

Example 5.5

$$\begin{array}{lclcl} (A, d) & \sqsubseteq & (A, \{\{X\}\}) & \sqsubseteq & (A, \{\{X, Y, Z\}\}) \\ (A, \{\{X\}\}) & \not\sqsubseteq & (A, \{\{X\}, \{Y\}\}) & \sqsubseteq & (A, \{\{X, Z\}, \{Y, W\}\}) \end{array}$$

Definition 3 (lub)

$$lub(\delta_1, \delta_2) = \{(X, Lub) | (X, SS_1) \in \delta_1, (X, SS_2) \in \delta_2\}$$

$$where \;\; Lub = \begin{cases} if & SS_1 \sqsubseteq SS_2 & then \;\; SS_2 \\ elseif & SS_2 \sqsubset SS_1 & then \;\; SS_1 \\ else & min(\{S_1 \cup S_2 | \forall S_1 \in SS_1, \forall S_2 \in SS_2\}) \end{cases} \Box$$

The set of all abstract constraints is a complete lattice w.r.t. $\sqsubseteq$, with top element $\top$. $\top$ contains no information, it approximates the whole set of admissible constraints. Let $\bot$ be a new symbol and let $\forall \delta \in Def$, $\bot \sqsubseteq \delta$ and $\neg \exists \delta \in Def$ such that $\delta \sqsubseteq \bot$. Thus $(Def \bigcup \{\bot\}, \sqsubseteq)$ is a complete lattice.

Definition 4 (Abstraction of a set of constraints)

$$\alpha(\Pi) = \cup_{\pi \in \Pi} \Delta(\pi) \;\; \Box$$

Definition 5 (Concretization of an abstract constraint)

$$\gamma(\Delta) = \{\pi \mid \pi \in Cons, \alpha(\pi) \sqsubseteq \Delta\} \Box$$

Once the abstract domain has been defined, let us show intuitively that it satisfies the conditions imposed in section 4.2. Conditions 3 to 6 are clear from the definition. Therefore we will concentrate on conditions 1 and 2.

Condition 1 requires that $\forall \delta_1, \delta_2 \in Def, \delta_1 \sqsubseteq \delta_2 \Rightarrow (\gamma(\delta_1) \Rightarrow \gamma(\delta_2))$. Let $trans(X, SS), (X, SS) \in \delta$ denote the constraint resulting from translating the element (X, SS) into the *Prop* domain. Then, $\forall (X, SS_2) \in \delta_2$:

- if $SS_2 = d, \exists(X, SS_1) \in \delta_1$ s.t. $SS_1 = d$. Thus $trans(X, SS_2) = trans(X, SS_1) = (X_i = 1)$ and it is clear that $trans(X, SS_1) \Rightarrow trans(X, SS_2)$
- if $SS_2 = \top$, then $trans(X, SS_2) = true$, thus $trans(X, SS_1) \Rightarrow trans(X, SS_2)$ for any $trans(X, SS_1)$
- if $SS_2 \in \wp(\wp(Pvar))$ then $trans(X, SS_2) = \{A_1 \wedge \cdots A_n \Rightarrow X | \forall S_2 \in SS_2, A_i \in S$. Let (X, SS_1) be the counterpart of (X, SS_2) in δ_1:
  - if $SS_1 = d$ it is clear that $trans(X, SS_1) \Rightarrow trans(X, SS_2)$
  - otherwise, $SS_1 \in \wp(\wp(Pvar)), \forall S_2 \in SS_2, \exists S_1 \in SS_1,\ s.t.\ S_1 \subseteq S_2$. Thus $\forall \pi_2 = \{A_1 \wedge \cdots A_n \Rightarrow X\} \in trans(X, SS_2), \exists \pi_1 = \{B_1 \wedge \cdots \wedge B_m \Rightarrow X\}, 1 \leq m < n$ s.t. $\{B_1, \cdots, B_m\} \subseteq \{A_1, \cdots, A_n\}$. Therefore, $trans(X, SS_1) \Rightarrow trans(X, SS_2)$

Condition 2 requires $\forall \delta_1, \delta_2 \in Def, \delta_1 \sqsubseteq lub(\delta_1, \delta_2), \delta_2 \sqsubseteq lub(\delta_1, \delta_2)$. From the definition of the *lub* function (definition 3), it is clear that condition 2 holds if $\forall(X, SS_1) \in \delta_1, \forall(X, SS_2) \in \delta_2$, either $SS_1 \sqsubseteq SS_2$ or $SS_2 \sqsubset SS_1$. Therefore, we only need to prove that, if the function $min(Temp), Temp = \{S_1 \cup S_2 | \forall S_1 \in SS_1, \forall S_2 \in SS_2\}$ is applied the resulting $\{(X, SS)\}$ satisfy that $\{(X, SS_1)\} \sqsubseteq \{(X, SS)\}$ and $\{(X, SS_2)\} \sqsubseteq \{(X, SS)\}$. It is clear, by definition of SS that $\{(X, SS_1)\} \sqsubseteq \{(X, Temp)\}$ and $\{(X, SS_2)\} \sqsubseteq \{(X, Temp)\}$. Intuitively, it obtains the lub of the constraints represented by each couple. The problem is that it may contain supersets, and this can be only due to the existence of a set of variables which appears in both SS_1 and SS_2. Therefore supersets can be eliminated preserving correctness by means of the function $min(Prod)$.

5.2 Abstract Conjunction Function

The abstract conjunction operation Λ is a function which takes as arguments two abstract constraints δ and δ', and returns the abstract constraint $\Lambda(\delta, \delta') = \Delta$. We will assume that $vars(\delta) = vars(\delta')$. If this is not the case, it is only necessary to add to δ the element $(X, \top)$ for each variable such that $X \in \delta', X \notin \delta$ or vice versa. Thus, let δ and δ' be two abstract constraints, such that $vars(\delta') = vars(\delta)$.

Definition 6 (Abstract conjunction function: $\Lambda(\delta, \delta')$)

$$\Lambda(\delta, \delta') = prop\_def(Different, \delta 1)$$

where

$$\begin{array}{ll} \delta 1 & = \{(X, SS1) | (X, SS') \in \delta', (X, SS) \in \delta, SS1 = add(SS, SS')\} \\ Different & = diff(\delta, \delta 1) \cup diff(\delta', \delta 1) \quad \square \end{array}$$

The function *add* takes as arguments two elements of Def and returns another element of Def which results from abstractly conjuncting the information contained in each element.

$$add(SS', SS) = \begin{cases} if & SS' \sqsubseteq SS & then \quad SS' \\ else\ if & SS \sqsubset SS' & then \quad SS \\ else & & min(SS \cup SS') \end{cases}$$

The function *diff* takes as arguments two abstract constraints δ and $\delta 1$ defined over the same program variables, and returns the elements $(X, SS1) \in \delta 1$ which are different from the correspondent $(X, SS) \in \delta$:

$$diff(\delta, \delta 1) = \{(Y, SS1) \in \delta 1 | \exists (Y, SS) \in \delta, SS \neq SS1\}$$

The function *prop_def* takes as arguments an abstract constraint $\delta 1$ and the set of elements $Elem = (X, SS1) \in \delta 1$ which have changed, and returns the abstract constraint $\delta 2$ resulting from propagating definiteness until fixpoint is reached. For reasons of efficiency, the set Elem should be ordered by definiteness, i.e. uniquely constrained variables should appear first:

$$prop\_def(Elem, \delta 1) = \begin{cases} if & Elem = \emptyset \quad then \quad \delta 1 \\ else & prop\_def(Elem1 \setminus (\{(X, SS2)\}), \delta 2) \end{cases}$$

$$\begin{array}{ll} where & \\ \delta 2 & = prop\_each((X, SS2), \delta 1) \\ Elem1 & = (Elem \setminus \{(Y, EE) | \exists (Y, SS) \in diff(\delta 1, \delta 2), EE \neq SS\}) \\ & \cup diff(\delta 1, \delta 2) \end{array}$$

Intuitively, the $\Lambda(\delta, \delta')$ function proceeds in three steps. First it obtains δ_1, which abstracts the constraint satisfaction of the conjunction of the constraints represented in the *Prop* domain, but before performing a "real" simplification of the resulting constraint. It is straightforward to see that function the *add* correctly abstracts this operation since it always takes the (minimized) union of the constraints, except for those cases in which a set of constraints for a variable X is known to be greater or equal (in the pre-order) than the set of constraints for X in the other abstract constraint. Second, it obtains those elements $(X, SS_1) \in \delta_1$ for which $\exists (X, SS) \in \delta$ s.t. $SS \neq SS_1$ or $\exists (X, SS') \in \delta'$ s.t. $SS' \neq SS_1$. The last step consists of recursively propagating the information for each element which has changed until fixpoint is reached.

Example 5.6 These functions allow us to accurately propagate definiteness in performing abstract conjunction. Consider the abstract constraints:

δ	$\{(X, \{\{Y\}, \{Z\}\}), (Y, \{\{Z\}\}), (Z, \top), (W, \top)\}$
$delta'$	$\{(X, \top), (Y, \{\{W\}\}), (Z, d), (W, \{\{Y\}\})\}$

We will first perform $prop\_def(Elem, \delta 1)$, where:

$$\begin{array}{lll} \delta 1 & = & add\_inf(\delta, \delta') = \{(X, \{\{Y\}, \{Z\}\}), (Y, \{\{Z\}, \{W\}\}), (Z, d), (W, \{\{Y\}\})\}, \\ Elem & = & diff(\delta, \delta 1) \cup diff(\delta', \delta 1) = \{(Z, d), (Y, \{\{Z\}, \{W\}\}), (W, \{\{Y\}\})\} \end{array}$$

- $Elem \neq \emptyset$, thus we execute $prop\_def(Elem1 \setminus \{(Z, d)\}, \delta 2)$, where

$$\begin{array}{lll} \delta 2 & = & prop\_each((Z, d), \delta 1) = \{(Z, d)\} \cup \{(X, d), (Y, d)\}, (W, \{\{Y\}\})\}, \\ diff(\delta 1, \delta 2) & = & \{(X, d), (Y, d)\}, \text{ and} \\ Elem1 & = & \{(X, d), (Y, d), (Z, d), (W, \{\{Y\}\})\} \end{array}$$

- $Elem1 \setminus \{(Z, d)\} \neq \emptyset$, execute $prop\_def(Elem2 \setminus \{(X, d)\}, \delta 3)$, where

$$\begin{array}{lcl}
\delta 3 & = & \mathit{prop\_each}((X,d),\delta 2) = \{(X,d)\} \cup \{(Y,d)\}),(Z,d),(W,\{\{Y\}\})\}, \\
\mathit{diff}(\delta 2,\delta 3) & = & \emptyset, \text{ and} \\
\mathit{Elem2} & = & \{(X,d),(Y,d),(W,\{\{Y\}\})\}
\end{array}$$

• $\mathit{Elem2} \setminus \{(X,d)\} \neq \emptyset$, execute $\mathit{prop\_def}(\mathit{Elem3} \setminus \{(Y,d)\},\delta 4)$, where

$$\begin{array}{lcl}
\delta 4 & = & \mathit{prop\_each}((Y,d),\delta 3) = \{(Y,d)\} \cup \{(X,d),(Z,d),(W,d)\}, \\
\mathit{diff}(\delta 3,\delta 4) & = & \{(W,d)\}, \text{ and} \\
\mathit{Elem3} & = & (\{(Y,d),(W,\{\{Y\}\})\} \setminus \{(W,\{\{Y\}\})\}) \cup \{(W,d)\} = \{(Y,d),(W,d)\}
\end{array}$$

• $\mathit{Elem3} \setminus \{(Y,d)\} \neq \emptyset$, execute $\mathit{prop\_def}(\mathit{Elem4} \setminus \{(W,d)\},\delta 5)$, where

$$\begin{array}{lcl}
\delta 5 & = & \mathit{prop\_each}((W,d),\delta 4) = \{(W,d)\} \cup \{(X,d),(Y,d),(Z,d)\}, \\
\mathit{diff}(\delta 4,\delta 5) & = & \emptyset, \text{ and} \\
\mathit{Elem4} & = & \{(W,d)\}
\end{array}$$

• $\mathit{Elem4} \setminus \{(W,d)\} = \emptyset$, thus fixpoint has been reached for $\delta 5 = \{(X,d),(Y,d),(Z,d),(W,d)\}$,

Consider now the abstract constraints:

δ	$\{(X,\{\{Y\},\{Z,W\}\}),(Y,\{\{Z,W\}\}),(Z,\top),(W,\top)\}$
δ'	$\{(X,\top),\{(Y,\{\{Z\}\}),(Z,\{\{Y\}\}),(W,\top)\}\}$.

The result of the execution of $\mathit{prop\_def}(\mathit{Elem},\delta 1)$, where

$$\begin{array}{ll}
\delta 1 = & \{(X,\{\{Y\},\{Z,W\}\}),(Y,\{\{Z\}\}),(Z,\{\{Y\}\}),(W,\top)\}, \text{ and} \\
\mathit{Elem} = & \{(Y,\{\{Z\}\}),(Z,\{\{Y\}\})\}
\end{array}$$

will be $\Delta = \{(X,\{\{Y\},\{Z\}\}),(Y,\{\{Z\}\}),(Z,\{\{Y\}\}),(W,\top)\}$, where fixpoint is reached after four iterations.

5.3 Other domain dependent functions needed

Let us define the rest of the abstract functions required by the framework.

Definition 7 (abstract_project)

$$\mathit{abstract\_project}(A_i,\delta) = \{(X,SS) | \forall (X,SS') \in \delta, X \in \mathit{vars}(A_i)\}$$

$$\mathit{where} \;\; SS = \begin{cases} \mathit{if} & SS' \in \{d,\top\} & \mathit{then} \;\; SS' \\ \mathit{elseif} & \mathit{restrict}(\mathit{vars}(A_i),SS') = \emptyset & \mathit{then} \;\; \top \\ \mathit{else} & & \mathit{restrict}(\mathit{vars}(A_i),SS') \end{cases}$$

Definition 8 (abstract_procedure_entry)

$$\mathit{abstract\_procedure\_entry}(C,A_i,\delta_{proj}) = \mathit{BodyV} \cup \mathit{abstract\_project}(H,\delta')$$

$$\begin{array}{lll}
\mathit{where} & \mathit{BodyV} & = \{(X,\top) | X \in \{\mathit{vars}(B) \setminus \mathit{vars}(H)\}\} \\
\mathit{and} & \delta' & = \mathit{rec\_abs\_conj}(\delta \cup \{(X,\top) | X \in \mathit{vars}(H)\}, A_i, C, 0, N) \\
\mathit{and} & \multicolumn{2}{l}{N \; \mathit{is\ the\ number\ of\ arguments\ of\ both} \; A_i \; \mathit{and} \; C \;\; \square}
\end{array}$$

The *rec_abs_conj* function applies recursively the abstract conjunction function to an abstract constraint δ and the abstraction of the constraint $X = Y$, in which X and Y are the i-arguments of A_i and C respectively.

Definition 9 (rec_abs_conj)

$$rec\_abs\_conj(\delta, A_i, C, N, N1) = \begin{cases} if & N = N1 \quad then \quad \delta \\ else & rec\_abs\_conj(\delta', A_i, C, N+1, N1) \end{cases}$$

$$\begin{array}{lll} where & \delta' & = \Lambda(\delta, \delta_{N+1}) \\ and & \delta_{N+1} & = \Delta(X = Y) \\ and & \multicolumn{2}{l}{X \text{ and } Y \text{ are the } N+1 \text{ arguments of } A_i \text{ and } C \text{ respectively}} \end{array}$$

Definition 10 (abstract_procedure_exit)

$$abstract\_procedure\_extend(C, A_i, \delta_{proj}, \delta_{ans}) =$$

$$abstract\_project(A_i, \Lambda(\delta_{proj}, \Lambda(abstract\_project(H, \delta_{ans}), \Delta(A_i = H))))$$

Definition 11 (abstract_procedure_extend)

$$abstract\_procedure\_extend(A_i, \delta_i, \delta_{exit}) = \Lambda(\delta_i, \delta_{exit})$$

Definition 12 (abstract_procedure_constraint)

$$abstract\_procedure\_constraint(A_i, \delta_{proj}) = \Lambda(\Delta(A_i), \delta_{proj})$$

Since all functions are defined in terms of the abstract conjunction and the abstract projection functions, the conditions required for their correctness (given in section 4.3), are satisfied if the abstract projection and the abstract conjunction functions are correct. The proof of the latter was sketched in the previous section. Therefore, we will focus on the former. Let δ be an abstract constraint, A_i be a literal and $\delta_{proj} = abstract\_project(A_i, \delta)$. We have to prove that $\exists \pi \in \gamma(\delta) \Rightarrow \overline{\exists}_{vars(A_i)} \pi \in \gamma(\delta_{proj})$. Let $(X, SS) \in \delta$. If $X \notin vars(A_i)$ it is clear that it has to be eliminated, as it done by the function. If $X \in vars(A_i)$ then:

- if $SS \in \{d, \top\}$ it is clear that all concrete constraints represented by δ will be represented also by δ_{proj} if (X, SS) remains unchanged.

- if $SS \in \wp(\wp(Var))$ and $restrict(vars(A_i), SS') = \emptyset$ it implies that we do not have information about the groundness characteristics of X in terms of a subset of variables in $vars(A_i)$. In other words, $\neg\exists(A_1 \wedge \cdots A_n \rightarrow X) \in trans(X, SS)$ s.t. $\{A_1, \cdots, A_n\} \subseteq vars(A_i)$. Therefore, the projection must be $\top$.

- Otherwise, $\forall(A_1 \wedge \cdots A_n \rightarrow X) \in trans(X, SS)$ s.t. $\{A_1, \cdots, A_n\} \subseteq vars(A_i)$, it is abstracted in δ_{proj}.

Program	Description	Cl	Var
vecadd	Adds two vectors	2	5
mortgage	The well known mortgage program	2	12
matvec	Multiplies two vectors	4	11
matmul	Multiplies two matrices	6	21
determ	Computes the determinant of a matrix	11	49
num	Number to letters-phonems translation (E.Vetillard)	97	233
motor-model	Motor modelization (W. Krautter)	53	364

Table 1: Benchmark characteristics

query	abstract call	abstract answer	Time (s)
vecadd(X,Y,Z)	X,Y	X,Y,Z	0.33
	X,Z	X,Y,Z	0.33
mortgage(P,T,I,R,B)	P,I,R	P,T,I,R,B	0.06
	P,R	P,T,R	0.14
matvec(X,Y,Z)	X,Y	X,Y,Z	0.36
	X,Z	X,Z	0.63
matmul(X,Y,Z)	X,Y	X,Y,Z	0.41
	X,Z	X,Z	1.18
determ(X,Y)	X	X,Y	0.65
nombre(X,Y,Z)	X,Y	X,Y,Z	5.01
	[ ]	X,Y,Z	5.06
(motor-model) run1	[ ]	[ ]	15.09
run2	[ ]	[ ]	25.96
run3	[ ]	[ ]	27.69

Table 2: Analysis results and timings

6 Implementation results

The analysis described in the previous sections has been implemented within our abstract interpretation framework and its implementation in Prolog, PLAI, which have thus been extended to perform analyses of CLP programs. This framework is based on that of Bruynooghe [1], optimized with the specialized domain-independent fixpoint defined in [17], extended to treat (and take advantage of) programs in non-normalized form, and, as mentioned before, generalized to support analysis of practical CLP languages following the guidelines presented in this paper.

It is important to note that the only modification that was needed for extending the framework itself was the addition of a clause which handles the case in which a literal is a constraint and treating syntactic differences among the set of CLP languages analyzed (which currently includes CLP(R), Prolog-III, and, of course, Prolog). Naturally, the domain-dependent abstract functions had to be implemented and incorporated into the system but almost all the existing implementation was reused. We believe that this supports our claim regarding the practical usefulness of the approach, specially considering that the resulting system can analyze reasonably sized programs in quite reasonable times.

Table 1 describes the programs analyzed, their function, the original size measured in number of clauses, and the total number of program variables, the latter two being relevant when considering the analysis times and information inferred. Table 2 presents the information inferred *at the query level* for each program and the analysis times in seconds (SparcStation IPC, Sicstus 2.1, compact code). The analysis times show that the analyzer can handle reasonable programs, even more if one takes into account that the abstract functions has been implemented in a rather naive way, and that the programs are highly recursive and/or have a large number of variables (43 just for the entry point, in the case of the motor example).

Regarding accuracy, the PLAI framework provides information at all points of the program, i.e. it provides not only the call and success patterns for each program predicate used in solving the query, but also the state of the information at each point of each program clause (and it keeps internally different predicate versions so that it can optionally perform program specializations). Therefore it is possible to observe how the execution of each subgoal affects the inferred information. However, due to lack of space and as mentioned above we only provide the results for the query variables. First, let us point out that for these benchmarks, whenever the abstract answer given in the table uniquely defines all variables in the query, the output abstract constraint (what has been called δ_{out}) *of each program clause* also implies that all variables in the clause are uniquely defined. Thus in these cases the analysis can obtain quite accurate information for most of the program points. The abstract calls and answers for the motor-model program (the last in the table) appear empty since the arity of the three different entry points is 0. While `run1` infers groundness for all program variables, `run2` and `run3` infers groundness for most of them but not for all. In general, and as would be expected, the analyzer accurately infers definiteness information when definiteness is explicit or can be propagated, but is (safely) inaccurate when it is the result of solving a system (as in the second abstract call for both `matvec` and `matmul`) or nonlinear constraints appear. Arguably, the results are quite acceptable for an analysis which is not specialized for any particular constraint system and is quite simple to implement.

7 Conclusions and Future Work

We have presented a practical approach to the dataflow analysis of programs written in constraint logic programming (CLP) languages using abstract interpretation. We have shown that, from the framework point of view, it suffices with quite simple extensions of traditional analysis methods, with the advantage that such methods have already been proved useful and practical and that efficient fixpoint algorithms have already been developed for them. Along this line we have proposed a simple but quite general generalization to the analysis of CLP programs of Bruynooghe's traditional framework CLP. As an example of the application of this approach, a complete, constraint system independent, abstract analysis has been presented for approximating definiteness information which is of quite general applicability since it uses in its implementation only constraints over the Herbrand domain. We have also presented some results from an implementation of this analysis which show that the approach is indeed practical and can be applied to applications written in languages such as CLP(R) and Prolog-III.

Proposals for future work include applying the proposed approach to other frameworks for analysis of LP and developing new abstractions within the framework proposed. Along this line, colleagues at K.U. Leuven have recently proposed an abstraction for freeness for linear arithmetic constraints and Herbrand equalities which uses our proposed extended version of Bruynooghe's framework and is implemented using PLAI [8]. Further collaborative work is in progress with K.U. Leuven combining the definiteness and freeness abstractions [9]. Our approach has also been applied to the extension of the GAIA framework [3], which is also closely related to Bruynooghe's and includes fixpoint extensions similar to those of PLAI, to the analysis of CLP programs [13]. Finally, we would like to note that the example definiteness analysis proposed as an example can be used to infer several other properties which have similar characteristics to definiteness, such, for example, its simple subclasses (e.g. "integer", "atomic", "constant", "numeric", etc.).

Acknowledgments

This paper has benefited from discussions with W. Winsborough, with partners in the PRINCE ESPRIT project, specially M. Bruynooghe, M. Codish, G. Janssens, A. DeWaal, and J. Gallagher, and with other members of the CLIP group at UPM, and from the observations of V. Saraswat and the anonymous referees, all of which the authors would like to thank. This work was funded in part by ESPRIT projects 5246 "PRINCE" and 7195 "ACCLAIM" and by the corresponding CICYT projects TIC91-0106-CE and TIC93-0975-CE.

References

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

[2] M. Bruynooghe and G. Janssens. Towards a framework for the abstract interpretation of constraint logic programs. In *Workshop on Logic Program Synthesis and Transformation*, LNCS. Springer-Verlag, 1992.

[3] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. In *Fourth IEEE Int. Conf. on Computer Languages (ICCL'92)*, April 1992.

[4] P. Codognet and G. Filé. Computations, Abstractions and Constraints in Logic Programs. In *Fourth Int. Conf. on Programming Languages*, April 1992.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[6] S. Debray, editor. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, volume 13(1–2). North-Holland, July 1992.

[7] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

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

[9] V. Dumortier, G. Janssens, W. Simoens, and M. Garcia de la Banda. Combining a Definiteness and a Freeness Abstraction for CLP Languages. In *Workshop on Logic Program Synthesis and Transformation*. Springer-Verlag, 1993. To appear in LNCS.

[10] M. Hanus. Analysis of Nonlinear Constraints in CLP(R). In *Tenth Int. Conf. on Logic Programming*, pages 83–99. MIT Press, June 1993.

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

[12] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

[13] G. Janssens and W. Simoens. On the Implementation of Abstract Interpretation Systems for (Constraint) Logic Programs. In *Workshop on Logic Program Synthesis and Transformation.* Springer-Verlag, 1993. To appear in LNCS.

[14] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. *Information Processing*, pages 601–606, April 1989.

[15] K. Marriott and H. Søndergaard. Analysis of Constraint Logic Programs. In *1990 North American Conf. on Logic Programming*, pages 531–547. MIT Press, 1990.

[16] C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third Int. Conf. on Logic Programming*, number 225 in LNCS, pages 463–475. Springer-Verlag, July 1986.

[17] 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.

[18] P. Van Roy and A. M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *North American Conf. on Logic Programming*, pages 501–515. MIT Press, October 1990.

[19] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth Int. Conf. and Symp. on Logic Programming*, pages 684–699, August 1988. MIT Press.