Global Analysis of Constraint Logic Programs

M. GARCIA DE LA BANDA and M. HERMENEGILDO
Universidad Politécnica de Madrid
and
M. BRUYNOOGHE, V. DUMORTIER, G. JANSSENS, and W. SIMOENS
Katholieke Universiteit Leuven

This article presents and illustrates a practical approach to the dataflow analysis of constraint logic programming 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 exist. This is shown by proposing a simple extension of Bruynooghe's traditional framework which allows it to analyze constraint logic programs. Then, and using this generalized framework, two abstract domains and their required abstract functions are presented: the first abstract domain approximates definiteness information and the second one freeness. Finally, an approach for combining those domains is proposed. The two domains and their combination have been implemented and used in the analysis of CLP($\Re$) and Prolog-III applications. Results from this implementation showing its performance and accuracy are also presented.

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; D.1.m [**Programming Techniques**]: Miscellaneous—*constraint logic programming*; D.3.2 [**Programming Languages**]: Language Classifications—*nonprocedural languages*; D.3.4 [**Programming Languages**]: Processors—*compilers*; F.3.1 [**Logics and Meaning of Programs**]: Specifying, Verifying and Reasoning about Programs

General Terms: Languages

Additional Key Words and Phrases: Abstract interpretation, constraint logic programming, global program analysis, program analysis

1. INTRODUCTION

The constraint logic programming (CLP) paradigm [Jaffar and Lassez 1987] is a relatively recent proposal which has emerged as the natural combination of the constraint solving and logic programming paradigms. This combination enhances

This research has been funded in part by the CEC through ESPRIT project 5246 PRINCE, by the Belgium National Fund for Scientific Research, by CICYT project TIC91-0106-CE, and by HCM project CHRX-CT94-0624 (ABILE).
Authors' addresses: M. García de la Banda and M. Hermenegildo, Universidad Politécnica de Madrid, Facultad de Informática, 28660-Boadilla del Monte, Madrid, Spain; email: {maria; herme}@fi.upm.es; M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens, Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, 3001 Heverlee, Belgium; email: {maurice; veroniek; gerda; wims}@cs.kuleuven.ac.be.

the flexibility and expressiveness of conventional logic languages. In this context, traditional logic programming (LP) can be seen as an instance of CLP in which constraints are equations over terms and in which the constraint solving is done by the well-known unification algorithm.

One of the main advantages of CLP languages is that they allow the programmer to specify the problem in a short, simple, and declarative way by means of high-level constraints, leaving the details of how these constraints are to be solved to the underlying constraint solver. When the execution of the program requires the full capabilities of the solver, the resulting efficiency is often quite good, in the sense that it would only be achievable in another language after an extensive and tedious programming effort. However, in the cases in which a simpler solver would suffice, the expressive power is paid in terms of efficiency. As it has recently been shown, efficiency can be recovered by performing several compile-time optimizations, mainly aimed at automatically specializing the program in order to reduce as much as possible the use of the general solver [Dumortier 1994; García de la Banda 1994; Jaffar and Maher 1994; Jaffar et al. 1992; Jørgensen et al. 1991; Marriott and Stuckey 1993; Marriott et al. 1994]. The significant speedups promised by these optimizations, and the fact that they need quite accurate compile-time information regarding the characteristics of the program, have motivated a growing interest in dataflow analysis of CLP languages and, in particular, in the application of abstract interpretation [Cousot and Cousot 1977].

Much work has been done using the abstract interpretation technique in the context of LP (e.g., Mellish [1986], Debray [1989], Bruynooghe [1991], Marriott and Søndergaard [1989], and Debray [1992b]). A number of systems have been built, some of which have shown the potential usefulness and practicality of this technique [Bueno et al. 1994; Debray 1992b; Le Charlier and Van Hentenryck 1994; Muthukumar and Hermenegildo 1992; Van Roy and Despain 1992; Warren et al. 1988]. Thus, it is natural to expect that this technique should also be useful in the context of CLP. A few general frameworks have already been defined for this purpose [Bruynooghe and Janssens 1992; Codognet and Filé 1992; Giacobazzi et al. 1993; Marriott and Søndergaard 1990]. However, one common characteristic of these frameworks is that they are either not implementation oriented or depart from the approaches that have been so far quite successful in the analysis of traditional logic programming (LP) languages.

This article shows how some of the LP-based techniques already developed and implemented can relatively easily be extended to the analysis of CLP programs. This point is illustrated by proposing a simple but quite powerful extension of Bruynooghe's traditional framework in order to make it applicable to the analysis of CLP programs. We also extend the framework to deal with passive constraints. Finally, we give correctness conditions for the resulting framework. The generalized description represents a fully specified algorithm for analysis of CLP programs.

Then, and using this generalized framework, two abstract domains and their required abstract functions are described. The abstract domain $Cons^{\mathcal{D}}$ determines whether program variables are *definite*, i.e., constrained to a unique value. The abstract domain $Cons^{\mathcal{F}^m}$ determines whether program variables are *free*, i.e., whether they can still take any possible value (at least according to their type, e.g., a variable that is constrained to be numerical but still ranges over the complete domain of numbers is considered as free). Finally, an approach for combining those domains is proposed. The idea is to use the definiteness information provided by $Cons^{\mathcal{D}}$ to obtain a more compact and efficient freeness abstraction while maintaining the precision of the original freeness abstraction. This combination leads to a *full* mode inference system which is, to the authors' knowledge, the first full mode system proposed for CLP.

The two abstract domains and their combination have been implemented within the abstract interpretation system PLAI [Muthukumar and Hermenegildo 1990; 1992]. This system is based on the framework of Bruynooghe [1991], optimized with the specialized domain-independent fixpoint defined in Muthukumar and Hermenegildo [1992] and generalized to support analysis of practical CLP languages, following the guidelines presented in this article. Results from this implementation showing its performance and accuracy are also presented.

Parts of the work in this article have already been presented previously. The generalization of abstract interpretation of LP toward CLP has been discussed in García de la Banda and Hermenegildo [1993] and Bruynooghe and Boulanger [1994]. A description of the definiteness analysis can be found in García de la Banda and Hermenegildo [1993] and García de la Banda [1994]. The freeness analysis and its optimizations have been described in Dumortier et al. [1993], Dumortier and Janssens [1994], and Dumortier [1994]. Dumortier and Janssens [1994] also explain how definiteness information can be exploited in order to improve the freeness abstraction.

2. BACKGROUND AND NOTATION

In this section we present some basic concepts of constraint logic programming and abstract interpretation, as well as the notation which will be used throughout the article. In doing this, we will follow mainly Jaffar and Lassez [1987], Jaffar and Maher [1994], and Cousot and Cousot [1977].

2.1 Constraint Domains and Programs

First we introduce some notational conventions. Uppercase letters generally denote collections of objects, while lowercase letters generally denote individual objects. u, v, w, x, y, z will denote variables; t will denote a term; p, q will denote predicate symbols; f will denote a function symbol; a, h will denote atoms; c will denote a constraint; ϵ will denote the empty constraint; b, g will denote an atom or a constraint; ρ will denote a rule; P, Q will denote programs; and B, G will denote goals, i.e., sequences of atoms and constraints. These symbols may be subscripted. $\tilde{x}$ denotes a sequence of distinct variables and, by abuse of notation, also the corresponding set of variables. $vars(o)$ denotes the set of variables occurring in the syntactic object o. Finally, $\exists_{-\tilde{x}}\phi$ denotes the existential closure of the formula ϕ except for the variables $\tilde{x}$, and $\tilde{\exists}\phi$ denotes the full existential closure of the formula ϕ.

As an example of a simple CLP program, consider the following, adapted from Jaffar and Maher [1994]: $sumto(x, y)$ expresses that y is the sum of the first x natural numbers.

$sumto(0, 0).$

$$sumto(x, y) \text{ :- } 1 \leq x, x \leq y, x' = x - 1, y' = y - x, sumto(x', y').$$

This simple program can be used to compute y from x (e.g., $sumto(3, y)$ returns $y = 6$ and then terminates), compute the x from y (e.g., $sumto(x, 15)$ returns $x = 5$ and then terminates), test whether a given x and y satisfy the relationship (e.g., $sumto(5, 15)$ succeeds and terminates and $sumto(3, 15)$ fails), or to answer more complex queries like ?- $y \leq 3, sumto(x, y)$ which gives rise to three answers $(x = 0, y = 0)$, $(x = 1, y = 1)$, and $(x = 2, y = 3)$ and then terminates. A direct translation of the above program into Prolog would require transforming each arithmetic equality into the `is/2` Prolog builtin:

$$sumto(x, y) \text{ :- } 1 \leq x, x \leq y, x' \textit{ is } x - 1, y \textit{ is } y' + x, sumto(x', y').$$

However, since the Prolog arithmetic builtins $is/2$ and $\leq /2$ require their second and both arguments, respectively, to be bound to a numerical value at run-time, the Prolog program would only execute queries in which both input arguments are constrained to a unique numerical value, such as ?- $sumto(2, 3)$ and ?- $sumto(2, 5)$. Carefully rewriting the second rule as

$$sumto(x, y) \text{ :- } 1 \leq x, x' \textit{ is } x - 1, sumto(x', y'), y \textit{ is } y' + x, x \leq y.$$

will also allow executing queries in which the second input argument is an unconstrained variable, such as ?- $sumto(3, y)$. While less general than the CLP program, this Prolog program will execute quite fast, because it is performing only simple arithmetic operations. Note also that rewriting the second rule as

$$sumto(x, y) \text{ :- } sumto(x', y'), x \textit{ is } x' + 1, y \textit{ is } y' + x, 1 \leq x, x \leq y.$$

the query ?- $sumto(3, y)$ will produce the answer $y = 6$ but then go into an infinite loop. The same happens, after producing the three answers, for the query ?- $sumto(x, y), y \leq 3$. In summary, although the functionality of the simple and elegant CLP program can only be obtained in Prolog by a more complex case-by-case program, the resulting Prolog program would probably execute faster than its CLP counterpart.

The example illustrates the expressiveness of CLP programming but also the challenge for the implementors. Ideally, it would be desirable for CLP systems to be strict generalizations of LP systems, not only from a functional point of view, but also from a performance point of view, i.e., in our example the general CLP program should offer a performance comparable with that of Prolog for queries such as ?- $sumto(3, y)$. To achieve this, some form of program analysis and creation of code dedicated to queries in this class looks unavoidable. Furthermore, one would also like the program to perform as well as possible even when actual constraint solving is being performed by the program. As mentioned in the introduction, it has been shown that such optimizations often require information from global analysis.

In the example, we have a constraint domain that is based on the constants 0, 1, the function $+$, and the predicates $=, <, \leq$ (3 is syntactic sugar for $1 + 1 + 1$; $x' = x - 1$ is syntactic sugar for $x = x' + 1$). In general, constants, functions, and predicates make up the signature Σ underlying the constraint domain. The so-called Σ-structure $\mathcal{D}$ consists of a domain, e.g., the domain of the real numbers, and an interpretation of constants, functions, and predicates over this domain, e.g., the standard arithmetic of the reals. A *primitive constraint* such as $1 \leq x$

is built from a predicate in Σ and terms built from constants and functions in Σ and from variables. Using logical connectives and quantifiers, primitive constraints can be combined into expressions of a language $\mathcal{L}$, called *constraints*. The pair $(\mathcal{D}, \mathcal{L})$ defines the constraint domain. The interested reader should consult Jaffar and Maher [1994] for a more formal and more detailed account, as well as for the assumptions that are usually made about $(\mathcal{D}, \mathcal{L})$.

As in the example above, a CLP program is a collection of rules of the form h :- B, where h (the *head*) is an atom built from a predicate (not in Σ), and B (the *body*) is a sequence $b_1, \ldots, b_n$ of atoms (not in Σ) and constraints. A goal G is also any sequence of atoms and constraints.

In Jaffar and Maher [1994] four relevant operations on constraints are mentioned, the first one being almost obligatory in any implementation of a CLP language:

(1) *Consistency* or *satisfiability* of a constraint c: $\mathcal{D} \models \tilde{\exists} c$.
(2) *Implication* or *entailment* of a constraint c_1 by another constraint c_0: $\mathcal{D} \models c_0 \rightarrow c_1$.
(3) *Projection* of a constraint c onto variables $\tilde{x}$: $\mathcal{D} \models \exists_{-\tilde{x}} c$.
(4) Detection that, given a constraint c, there is only one value that a variable x can take that is consistent with c: $\mathcal{D} \models c(x_1, \tilde{y}) \wedge c(x_2, \tilde{y}) \rightarrow x_1 = x_2$. We say that x is *definite* in c and denote by $def(c)$ the set of definite variables in c.

2.2 CLP Operational Semantics

In Jaffar and Maher [1994], the interested reader can find a very general operational semantics which takes passive constraints into account, separates the generation of new constraints from the consistency check of the constraints accumulated in the constraint store, and is not tailored to any particular computation rule.

The work reported here concerns the analysis of programs under the widely used left-to-right computation rule (as in Prolog). In the first part of this article, we focus on programs without passive (i.e., delayed) constraints. The treatment of passive constraints is deferred to Section 5.3. Another assumption is that the considered systems are quick-checking [Jaffar and Maher 1994], i.e., the addition of new constraints is immediately followed by a consistency check of the constraint store.

Under these simplifications, the state of the computation can be described by a pair $\langle G; c\rangle$, where G is the sequence of constraints and atoms yet to be executed, and c is the constraint store containing the constraints accumulated so far. The operational behavior of a program can be described by a set of sequences of states (*SLD sequences*), each sequence starting with $\langle G; true\rangle$ where G is the query.

Such SLD sequences are manipulated by transitions whose behavior — given the left-to-right computation rule — is determined by the leftmost element in the goal of the last state in the sequence. Formally, an incomplete SLD sequence ending in a consistent state can be extended by the following transitions which are formulated as rewrite rules (S represents an SLD sequence, and :: is used to concatenate SLD sequences):

—$S :: \langle c', G; c\rangle$
$\stackrel{c}{\rightarrow} S :: \langle c', G; c\rangle :: \langle G; c' \wedge c\rangle$ if $consistent(c' \wedge c)$ or
$\stackrel{c}{\rightarrow} S :: \langle c', G; c\rangle :: \langle G; false\rangle$ if $inconsistent(c' \wedge c)$

—$S :: \langle a, G; c\rangle$ rewrites to a set of sequences, one for each rule of the program defining the predicate symbol of a. Let $\rho : h\text{:-}b_1, \ldots, b_n$ be such a (properly renamed) rule, then
$\xrightarrow{r} S :: \langle a, G; c\rangle \overset{\rho}{::} \langle b_1, \ldots, b_n, G; a = h \wedge c\rangle$ if $consistent(a = h \wedge c)$[1] or
$\xrightarrow{r} S :: \langle a, G; c\rangle \overset{\rho}{::} \langle b_1, \ldots, b_n, G; \mathit{false}\rangle$ if $inconsistent(a = h \wedge c)$

Note that an SLD sequence is very similar to a partial SLD derivation in Lloyd [1987]. In particular, an SLD sequence represents a *complete* derivation if its last state

—contains a goal whose leftmost atom has a predicate symbol for which the program has no rules (a failed SLD derivation);
—has *false* in its constraint store (also a failed SLD derivation, but this case is distinguished from the previous one because some analyses can be interested in the question of at which points an inconsistent store can be introduced; such analyses will use abstractions that can distinguish an inconsistent store from consistent ones);
—contains an empty goal (a successful SLD derivation); the constraint store then provides the answer.[2]

Ignoring the search rule (see Le Charlier et al. [1994] for a framework taking the search rule into account), the operational semantics is given by the fixpoint of the operator which applies the above transitions on incomplete SLD sequences, starting from the initial sequence $\langle G; true\rangle$. (Alternatively, the set of all complete sequences can be collected in an SLD tree as in Lloyd [1987]). The fixpoint of the operator — a set of complete SLD sequences — represents the operational semantics as it describes in sufficient detail the behavior of the program for the analyses considered in this article. If desired, sequences could be instrumented with more detail (e.g., Mulkers et al. [1994]).

2.3 Abstract Interpretation

The most familiar framework for abstract interpretation is defined in terms of Galois connections and Galois insertions [Cousot and Cousot 1977; 1992a].

Definition 2.3.1 (*Galois Connection*). A Galois connection is a quadruple $(Dom^{\mathcal{C}}, \alpha, Dom^{\mathcal{A}}, \gamma)$ where:

(1) $(Dom^{\mathcal{C}}, \leq^{\mathcal{C}})$ and $(Dom^{\mathcal{A}}, \leq^{\mathcal{A}})$ are posets called *concrete* and *abstract domains* respectively;
(2) $\alpha : Dom^{\mathcal{C}} \rightarrow Dom^{\mathcal{A}}$ and $\gamma : Dom^{\mathcal{A}} \rightarrow Dom^{\mathcal{C}}$ are functions called *abstraction* and *concretization functions* respectively, satisfying that for every $d^{\mathcal{A}} \in Dom^{\mathcal{A}}$ and $d^{\mathcal{C}} \in Dom^{\mathcal{C}}$, $\alpha(d^{\mathcal{C}}) \leq^{\mathcal{A}} d^{\mathcal{A}}$ iff $d^{\mathcal{C}} \leq^{\mathcal{C}} \gamma(d^{\mathcal{A}})$.

Definition 2.3.2 (*Galois Insertion*). A Galois insertion is a Galois connection satisfying $\alpha(\gamma(d^{\mathcal{A}})) = d^{\mathcal{A}}$.

[1]The label ρ on :: identifies the renamed rule used in solving a. The expression $a = h$ is an abbreviation for the conjunction of the corresponding primitive equations.
[2]The answer can be conditional in case one allows passive constraints, as then the store may be inconsistent.

The Galois connection corresponds to a perfect situation where each concrete property has a unique best abstract approximation. Thus, only one of $\{\alpha, \gamma\}$ needs to be specified, since if one exists the other is determined by the properties of the definition. In addition a Galois insertion has no superfluous elements in the abstract domain. The following specifies the notion of approximation (in terms of γ) which is then extended from the primitive domains to function domains:

Definition 2.3.3 (*Approximation*). Let $(Dom^{\mathcal{C}}, \alpha, Dom^{\mathcal{A}}, \gamma)$ be a Galois insertion, and let $\mu^{\mathcal{C}} : Dom^{\mathcal{C}} \rightarrow Dom^{\mathcal{C}}$ and $\mu^{\mathcal{A}} : Dom^{\mathcal{A}} \rightarrow Dom^{\mathcal{A}}$ be monotonic functions. We say that $d^{\mathcal{A}} \in Dom^{\mathcal{A}}$ *γ-approximates* $d^{\mathcal{C}} \in Dom^{\mathcal{C}}$, denoted $d^{\mathcal{A}} \propto_\gamma d^{\mathcal{C}}$, if $d^{\mathcal{C}} \leq^{\mathcal{C}} \gamma(d^{\mathcal{A}})$. We say that $\mu^{\mathcal{A}}$ *γ-approximates* $\mu^{\mathcal{C}}$, denoted $\mu^{\mathcal{A}} \propto_\gamma \mu^{\mathcal{C}}$, if for every $d^{\mathcal{A}} \in Dom^{\mathcal{A}}$, $d^{\mathcal{C}} \in Dom^{\mathcal{C}}$ such that $d^{\mathcal{A}} \propto_\gamma d^{\mathcal{C}}$ then $\mu^{\mathcal{A}}(d^{\mathcal{A}}) \propto_\gamma \mu^{\mathcal{C}}(d^{\mathcal{C}})$.

As illustrated in Section 2.2, the information of interest about a program — in our case the operational semantics — can often be expressed as the least fixpoint of a function. Formally one writes $[\![P]\!] = lfp(\mu^{\mathcal{C}})$ where $\mu^{\mathcal{C}} : Dom^{\mathcal{C}} \rightarrow Dom^{\mathcal{C}}$ is a monotonic operator on the concrete domain $Dom^{\mathcal{C}}$ and where $[\![P]\!]$ expresses the meaning of the program. Such a formalization provides the foundation for an abstract interpretation of the program. By introducing an appropriate Galois insertion $(Dom^{\mathcal{C}}, \alpha, Dom^{\mathcal{A}}, \gamma)$ and defining a monotonic function $\mu^{\mathcal{A}} : Dom^{\mathcal{A}} \rightarrow Dom^{\mathcal{A}}$, which approximates $\mu^{\mathcal{C}}$ and whose fixpoint can be computed or approximated by a finite computation, one can obtain information about the least fixpoint of $\mu^{\mathcal{C}}$. This is expressed by the following result [Cousot and Cousot 1992a]:

THEOREM 2.3.4. *Let $(Dom^{\mathcal{C}}, \alpha, Dom^{\mathcal{A}}, \gamma)$ be a Galois insertion, and let $\mu^{\mathcal{C}} : Dom^{\mathcal{C}} \rightarrow Dom^{\mathcal{C}}$ and $\mu^{\mathcal{A}} : Dom^{\mathcal{A}} \rightarrow Dom^{\mathcal{A}}$ be monotonic functions such that $\mu^{\mathcal{A}} \propto_\gamma \mu^{\mathcal{C}}$. Then $lfp(\mu^{\mathcal{A}}) \propto_\gamma lfp(\mu^{\mathcal{C}})$.*

The construction of $\mu^{\mathcal{A}}$ often takes a systematic approach which involves replacing the basic operations in the concrete semantics operator $\mu^{\mathcal{C}}$ by the corresponding abstract operations in $\mu^{\mathcal{A}}$ (e.g., Cousot and Cousot [1992a] and Nielson [1988]). Given that the basic abstract operations approximate their concrete counterparts, it is generally straightforward to prove that $\mu^{\mathcal{A}}$ approximates $\mu^{\mathcal{C}}$.

3. TOWARD A CLP ANALYSIS FRAMEWORK

There has been considerable interest in developing new abstract interpretation frameworks for CLP languages. To these authors' knowledge, at least four frameworks have been proposed previously or simultaneously with our work.[3] Marriott and Sondergaard [1990] present a general and elegant semantics-based framework. It is based on a definition-independent metalanguage 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 [Bruynooghe 1991]

[3]The ideas illustrated in this article were first presented at the ICLP'91 Workshop on Constraint Logic Programming.

and the optimizations proposed for it [Muthukumar and Hermenegildo 1992]) 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 [1992] also present a quite general framework, for the description of both CLP languages and their static analyses, and an implementation approach. Although more concrete, their proposal is still more abstract than the level pointed out above as our objective. On the other hand they introduce the quite interesting idea of implementing the abstract functions actually using constraint solvers, to which we will return later.

Giacobazzi et al. [1993] formulate a general algebraic framework for constraint logic programming. They formulate the operational and fixpoint semantics within this framework and show that abstract interpretation is simply another instance of the general framework which safely approximates the instance given by the concrete constraint system. Also, their work is in fairly general terms and does not offer much to the application developer.

Finally, Bruynooghe and Janssens [1992] present a specialized framework (which was developed in parallel with the proposal presented in this article) 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 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 their arguments and on the particular constraint system.

The view of constraints as suspended goals could be interesting and worth pursuing. However, this makes it more difficult to make the framework fully general. We prefer to take the more traditional notion presented in the CLP scheme (as introduced in the previous sections) in which constraints take the place of substitutions and in which goals always either succeed or fail, in the former case possibly *placing* new constraints.[4]

One of the main points of this article is to show that standard abstract interpretation frameworks for logic programs are 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. Indeed, in this traditional view of CLP the role of goals and their control are 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.

[4]In fact, actual suspension, as is often used in the solving of nonlinear 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 [Hanus 1993; Marriott et al. 1994]. This issue is discussed further in Section 5.3.

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 graph, the implementation and optimizations of the fixpoint algorithm, the notions of projection and extension, etc. This has the advantage that the 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.

4. MODIFYING THE CLP OPERATIONAL SEMANTICS

The states appearing in the fixpoint of the concrete operational semantics are of the form $\langle g, G; c\rangle$ where c is a constraint store over an unbounded number of variables. A basic insight underlying the framework of Bruynooghe [1991] is that, when optimizing a particular predicate, most optimizations only need information about the variables in the clauses defining such predicate. Therefore, when analyzing g, the analysis is not interested in the properties of all program variables but only of the variables of the clause g belongs to. This information is collected by a slightly different operational semantics which is called LSLD (Local SLD) in Bruynooghe and Boulanger [1994]. In our constraint setting, we can rephrase LSLD as an operator on LSLD sequences as follows:

—The c-transition on $S :: \langle c', G; c\rangle$ is as before.

—The r-transition on $S :: \langle a, G; c\rangle$ for $consistent(a = h \wedge c)$ becomes:
$S :: \langle a, G; c\rangle \xrightarrow{r} S :: \langle a, G; c\rangle \overset{\rho}{::} \langle b_1, \ldots, b_n; \exists_{-vars(\rho)}(a = h \wedge c)\rangle$, where $\rho : h$:-$b_1, \ldots, b_n$.
The r-transition for $inconsistent(a = h \wedge c)$ is unmodified. Because the r-transition computes a constraint store over the variables of ρ, it is called the *entry transition* in the future.

—In addition, an *exit transition* is introduced for states where the goal is the empty left-over of the body of a (uniquely renamed) rule $\rho : h$:-$b_1, \ldots, b_n$ (denoted $\Box_\rho$). Note that the transition is not only based on the last state in the sequence, but also on the state prior to the application of the entry transition using ρ (marked by $\overset{\rho}{::}$):
$S_1 :: \langle a, G; c\rangle \overset{\rho}{::} \langle b_1, \ldots, b_n; c_{in}\rangle :: S_2 :: \langle \Box_\rho; c_{out}\rangle \xrightarrow{exit}$
$S_1 :: \langle a, G; c\rangle \overset{\rho}{::} \langle b_1, \ldots, b_n; c_{in}\rangle :: S_2 :: \langle \Box_\rho; c_{out}\rangle :: \langle G; \exists_{-vars(\rho_0)}(c \wedge a = h \wedge c_{out})\rangle$.

ρ_0 is the rule containing a, G as tail of its body. Note that, due to the renaming, there is a unique state $\langle a, G; c\rangle$ to which an entry transition using ρ was applied. Note also that there exists a constraint c_{new} such that $c_{out} = c_{in} \wedge c_{new}$.

As before, the initial sequence is $\langle G; true\rangle$, with G being the query, and the operational semantics is given by the fixpoint of the operator applying transitions on incomplete sequences. Though the exit transitions introduce extra states $\langle \Box_\rho; c_{out}\rangle$ in LSLD sequences, there is a strong equivalence between SLD sequences and LSLD sequences using the same renamed rules in the same order: for every state $\langle b_1, \ldots, b_n, G; c\rangle$ in an SLD sequence with $b_1, \ldots, b_n$ the tail of some renamed rule ρ, there is a state $\langle b_1, \ldots, b_n; \exists_{-vars(\rho)} c\rangle$ in the corresponding LSLD sequence. This can be proved by induction. Consequently, the fixpoint of the LSLD operator carries the same amount of relevant information (i.e., what are the properties of $vars(b_i)$ of a state $\langle b_i, \ldots; c\rangle$) as the fixpoint of the original SLD operator.

An SLD sequence can be represented by an AND tree (a proof tree, to be distinguished from an SLD tree which is a search tree). The children of the root are the atoms and constraints of the query. An atom a is paired with the head of the rule $\rho : h$:-$b_1, \ldots, b_n$ that is used by the entry transition on a (the sequence contains $\langle a, \ldots; \ldots\rangle \overset{\rho}{::} \langle b_1, \ldots, b_n; c_1\rangle$). The constraint store adorns the tree as shown in the fragment of Figure 1; c_i is the constraint store of the state $\langle b_i, \ldots, b_n; c_i\rangle$. It contains the information about the variables of b_i at the point where b_i is to be processed. As described in Bruynooghe [1991], the set of all AND trees, which represents the operational semantics of the program, can be collected in an AND-OR tree where nodes are adorned with sets of constraint stores (this gives the collecting semantics). Using a tabulation technique, repeated computations can be avoided: there is no point in collecting states which are renamings of each other; therefore, states are tabled with $\langle b_1, \ldots, b_n; c_1\rangle$ as *key* and the corresponding $\langle \Box_\rho; c_{out}\rangle$ as *answer*. A sequence ending in a tabled state is extended with an exit operation which uses the tabled answer, thus avoiding the construction of a renaming of an already existing subsequence. The LSLD semantics is thus transformed into the LSLDT semantics [Bruynooghe and Boulanger 1994].

Tabulation allows abstracting the AND-OR tree, representing the concrete collecting semantics, by an AND-OR graph. In practice, however, abstract interpretation systems such as PLAI [Muthukumar and Hermenegildo 1990; 1992] and GAIA [Englebert et al. 1992; Le Charlier and Van Hentenryck 1994; Le Charlier et al. 1991] are based on a variant of the above tabulation technique, where the stored key is not $\langle b_1, \ldots, b_n; c_1\rangle$. Instead, with $\langle a, G; c\rangle$ as the preceding state, $\langle a; \exists_{-vars(a)} c\rangle$ is stored as key and $\exists_{-vars(a)}(a = h \wedge c_{out})$ as answer for an atom a if $\langle \Box_\rho; c_{out}\rangle$ has occurred when resolving a with ρ. If a state $\langle a', G'; c'\rangle$ is met such that $\langle a'; \exists_{-vars(a')} c'\rangle$ is a renaming of $\langle a; \exists_{-vars(a)} c\rangle$, then no entry transition with a renaming of ρ is performed. Instead, the sequence is extended with a state $\langle G'; \exists_{-vars(\rho'_0)}(c' \wedge a' = a \wedge \exists_{-vars(a)}(a = h \wedge c_{out}))\rangle$ for each tabled answer $\exists_{-vars(a)}(a = h \wedge c_{out}))$ (table lookup transition), where ρ'_0 is the rule with a', G' as tail of its body, and $a' = a$ performs renaming. The advantage of this tabulation variant is to avoid an entry transition. However, some table lookups can be missed because different states $\langle a, \ldots; c\rangle$ can give rise to $\langle b_1, \ldots, b_n; c_{in}\rangle$ that are renamings of each other (and extra work will be done: a lookup transition for every atom b_i and a c-transition for every constraint b_i). With so-called *seminormalized* programs where calls have the form $p(x_1, \ldots, x_n)$ (all x_i different), the disadvantage disappears. With the heads also of the form $p(x_1, \ldots, x_n)$ (*normalized* programs), $\exists_{-vars(\rho)}(a = h \wedge c)$ and $\exists_{-vars(\rho_0)}(a = h \wedge c_{out})$, where h:-... is used to resolve a, reduce to simple renaming operations. The price for (semi-)normalization is that there are more constraints in rule bodies and, more importantly, more variables. The latter can have a significant effect in applications where the size of an element in the abstract domain can be exponential in the number of rule variables. Also, for some applications, (semi-)normalization may result in loss of precision.

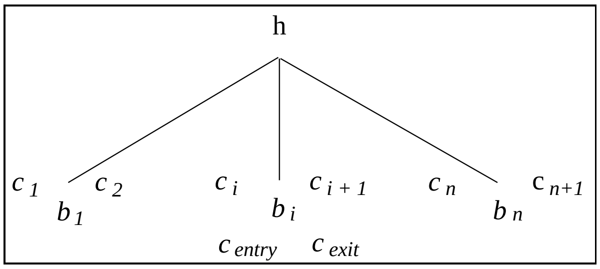

Fig. 1. Naming conventions for constraints.

Within the proposed LSLD semantics, it is convenient to name constraint stores differently, depending on the point in a rule to which they correspond. The same conventions will be used for the abstract constraint stores. Consider, for example, the rule h :- $b_1, \cdots, b_n$. Let c_i and c_{i+1} be the constraint stores to the left and right of the subgoal $b_i, 1 \leq i \leq n$ in this rule. See Figure 1.

—c_i and c_{i+1} are, respectively, the *call constraint* and the *success constraint* for b_i.

—c_1 and c_{n+1} are, respectively, the *in constraint* and the *out constraint* of the rule (also denoted by c_{in} and c_{out}). Note that c_1 and c_{n+1} are also the call constraint for b_1 and the success constraint for b_n, respectively.

—c_i projected over the variables of b_i is the *entry constraint* (represented by c_{entry}) of b_i, and the answer constraint $\exists_{-vars(b_i)}(b_i = h' \wedge c_{out})$ for b_i is called *exit constraint* (represented by c_{exit}). Note that these two constraints are defined over the variables in b_i, instead of over the variables of the rule.

5. EXTENSION OF THE ANALYSIS FRAMEWORK

As mentioned in the previous section, the framework of Bruynooghe [1991] provides an algorithm for safely abstracting an operational collecting semantics represented as an AND-OR tree by a finite AND-OR graph. The extension of the framework toward CLP is founded on the observation that the LSLD and LSLDT operational semantics are also valid for CLP. As a consequence, the extension replaces the set of substitutions adorning the AND-OR tree in the original framework by sets of constraint stores and replaces unification by conjunction. The algorithm is based on a number of primitive transitions which have to approximate transitions on states $\langle G; C\rangle$, where G is a sequence of constraints and atoms, and C is a set of constraint stores belonging to $Cons_{\tilde{x}}^{C}$ (denoting the set of all sets of constraint stores over the variables $\tilde{x}$). The abstract transitions operate on states $\langle G; AC\rangle$ with the abstract constraint AC, a description of a set of constraint stores, belonging to $Cons_{\tilde{x}}^{\mathcal{A}}$ (denoting the set of all descriptions of sets of constraint stores over $\tilde{x}$). The extension of the framework also involves a reformulation of the safety conditions of the primitive transitions in the constraint setting.

In some program points, the set of constraint stores C to be abstracted as AC is the fixpoint of a sequence $C_1 \subseteq C_2 \subseteq C_3 \subseteq \ldots$. Here, the standard theory of abstract interpretation comes in with the Galois insertion as the most popular approach for linking $Cons_{\tilde{x}}^{C}$ with $Cons_{\tilde{x}}^{\mathcal{A}}$ (see Marriott [1993] for others). The theory provides a method for safely approximating the fixpoint of the sequence $C_1 \subseteq C_2 \subseteq \ldots$. Having for each AC in the AND-OR graph a Galois insertion between $Cons_{\tilde{x}}^C$ and $Cons_{\tilde{x}}^A$, a Galois insertion is induced between the set of all AND-OR trees representing the collecting semantics and the set of all abstract AND-OR graphs.

5.1 The Abstract Domain

The elements to be abstracted in the collecting semantics are sets of constraint stores, a constraint store being built from primitive constraints through conjunction and projection. All constraint stores in the same set are over some set of variables $\tilde{x}$. Thus, the concrete domain is $(Cons_{\tilde{x}}^C, \leq^C)$ where $\leq^C$ is the subset relation. The concrete domain is a lattice whose minimal element is $\emptyset$ and whose maximal element is the set of all possible constraint stores over $\tilde{x}$. Whether *false* is also considered as a constraint store depends on the kind of analysis one is interested in.

The abstract domain $Cons_{\tilde{x}}^A$ consists of descriptions (denoted AC) and is equipped with an order relation $\leq^A$. Descriptions are given a meaning by the concretization function γ. For the analyses considered in this article, the meaning of descriptions are sets closed under equivalence. For example, if $x+y=2 \wedge x-y=0$ is in $\gamma(AC)$, then so will $x=1 \wedge y=1$.

A special class of descriptions are those where the represented sets are closed under *antientailment*: a description representing a constraint also represents all stronger constraints. Formally, if $c \in \gamma(AC)$ and $c' \rightarrow c$ then $c' \in \gamma(AC)$.[5] This class is the CLP counterpart of substitution-closed (downward-closed) descriptions in abstract interpretation of logic programs [Debray 1992a]. Such domains have the special property that, if AC is a valid description of the computation at state s_i in the collecting semantics, then it is also a valid description (though usually rather imprecise) of the state s_{i+1}. Indeed, the standard semantics can only strengthen the constraints by adding constraints to the store. The definiteness domain developed in Section 6 is such a domain. If a variable is constrained to a unique value by some constraint then it is certainly so under stronger constraints.

Another special class of descriptions represents sets closed under entailment (upward closed) (at least if unsatisfiable constraints are discarded): a description representing a constraint ($\neq$ *false*) represents also all weaker constraints, formally: if $c \in \gamma(AC)$ ($c \neq$ *false*) and $c \rightarrow c'$, then $c' \in \gamma(AC)$. The freeness domain developed in Section 7 is such a domain. If a variable can still take all possible values under some constraint, then it can do so under weaker constraints.

As stated in Section 2.3, the most familiar setting for abstract interpretation is the Galois insertion. The concretization function γ and the abstraction function α provide a tight linkage between the concrete and the abstract domain. As a consequence one can specify the safety conditions of the functions used in formulating the abstract semantics as well in terms of the concretization function as in terms of the abstraction function. As discussed in Marriott [1993], abstract interpretation has also been studied in settings with a weaker linkage between abstract and concrete domain. Here we follow the weaker setting of the original framework of Bruynooghe [1991] where only a concretization function γ is assumed. However, the formulation

[5]Note that each description that is closed under antientailment automatically represents the constraint *false*.

is modified in one aspect. To ensure termination for abstract domains allowing for infinite ascending chains $AC_1 <^{\mathcal{A}} AC_2 <^{\mathcal{A}} AC_3 <^{\mathcal{A}} \ldots$, the standard notion of a widening operator [Cousot and Cousot 1977; 1992b] is used.

Let $(Cons_{\tilde{x}}^{C}, \leq^{C})$ be the powerset of the set of all constraint stores, ordered by set inclusion. The minimal requirements on $(Cons_{\tilde{x}}^{\mathcal{A}}, \leq^{\mathcal{A}})$ are:

(1) A preorder $\leq^{\mathcal{A}}$ satisfying that $\forall AC_1, AC_2 \in Cons_{\tilde{x}}^{\mathcal{A}}$ if $AC_1 \leq^{\mathcal{A}} AC_2$ then $\gamma(AC_1) \subseteq \gamma(AC_2)$. The preorder allows to define an equivalence relation: $AC_1 \equiv^{\mathcal{A}} AC_2$ iff $AC_1 \leq^{\mathcal{A}} AC_2$ and $AC_2 \leq^{\mathcal{A}} AC_1$. The relation $\equiv^{\mathcal{A}}$ has the property $AC_1 \equiv^{\mathcal{A}} AC_2 \rightarrow \gamma(AC_1) = \gamma(AC_2)$.
(2) An upper bound operator $upp : Cons_{\tilde{x}}^{\mathcal{A}} \times Cons_{\tilde{x}}^{\mathcal{A}} \rightarrow Cons_{\tilde{x}}^{\mathcal{A}}$ such that $AC_i \leq^{\mathcal{A}} upp(AC_1, AC_2)$ $(i = 1, 2)$.
(3) A maximal element named $\top$ such that $\gamma(\top) =$ the set of all constraints over $\tilde{x}$ and $\forall AC \in Cons_{\tilde{x}}^{\mathcal{A}},\ AC \leq^{\mathcal{A}} \top$.
(4) A minimal element $\bot$ such that $\gamma(\bot) = \emptyset$ and $\forall AC \in Cons_{\tilde{x}}^{\mathcal{A}},\ \bot \leq^{\mathcal{A}} AC$.
(5) A widening operator $\mathcal{W} : Cons_{\tilde{x}}^{\mathcal{A}} \times Cons_{\tilde{x}}^{\mathcal{A}} \rightarrow Cons_{\tilde{x}}^{\mathcal{A}}$ such that $AC_i \leq^{\mathcal{A}} \mathcal{W}(AC_1, AC_2)(i = 1, 2)$ and such that there does not exist an infinite chain $ACa_1, ACb_1, ACa_2, ACb_2, ACa_3, \ldots$ such that, for all i, $not(ACb_i \leq^{\mathcal{A}} ACa_i)$ and for all $i > 1, ACa_i = \mathcal{W}(ACa_{i-1}, ACb_{i-1})$.

Condition (1) allows different descriptions that are not equivalent to represent the same set of constraints. However this is better avoided, as it can decrease precision and increase computation time. Condition (2) states that there must be an upper bound operator, i.e., that it must be possible to approximate two or more descriptions by a single one. Of course, it is desirable to define *upp* as precise as possible. With the abstract domain a complete partial order, the optimal *upp* is the least upper bound. Condition (3) implies the existence of a maximal element; it is a convention to name it $\top$. Condition (3) also states that it must represent the set of all constraints. This assures that every set of constraints has an abstraction. Condition (4) imposes a minimal element $\bot$ representing the empty set of constraints. This provides a precise abstraction for states in unreachable program points. Also, it provides the initial value for computing a fixpoint of a function over the abstract domain. Finally, condition (5) ensures existence of a widening operator which can enforce a safe approximation of a fixpoint in a finite number of steps ($ACa_1, ACa_2, \ldots$ are the successive approximations of the fixpoint, $ACb_1, ACb_2, \ldots$ the values resulting from the new iterations). Notice that *upp* can be used as widening operator in domains without infinite ascending chains.

5.2 The Abstract Operations

The algorithm computes an AND-OR graph adorned with abstract constraints (elements of the abstract domain). It also computes Table, an initially empty table, with elements of the form $(\langle a, AC_{entry}\rangle, AC_{exit})$. In these entries a is an atom, and AC_{entry} (the entry constraint) and AC_{exit} (the exit constraint) are abstract constraints over $vars(a)$. The pair $\langle a, AC_{entry}\rangle$ is the key of the table element, and AC_{exit} is the (current) answer for the call a with abstract entry constraint AC_{entry}. AC_{exit} is used by table lookups.[6] The graph is initialized with an AND

[6] A similar table is used in the concrete LSLDT semantics, but a key is then associated with a set of answers.

node having one child for each atom or constraint in the query ?-$g_1, \ldots, g_n$ and an abstract call constraint AC for g_1. This initialization represents the set of initial LSLDT sequences $\langle g_1, \ldots, g_n; c\rangle$ where $c \in \gamma(AC)$. The algorithm builds a complete graph by applying transitions in a controlled way.

Below, we make use of abstract projection, denoted $\exists^{\mathcal{A}}_{-\tilde{x}}$, and abstract conjunction $\wedge^{\mathcal{A}}$. They are intended to approximate projection and conjunction respectively. More formally:

—The abstract projection $\exists^{\mathcal{A}}_{-\tilde{x}}$ is a safe approximation of the concrete projection if for any constraint c and for any abstract constraint AC such that $c \in \gamma(AC)$ it holds that $\exists_{-\tilde{x}} c \in \gamma(\exists^{\mathcal{A}}_{-\tilde{x}} AC)$.

—The abstract conjunction $\wedge^{\mathcal{A}}$ is a safe approximation of the concrete conjunction if for any two constraints c_1, c_2 and for any two abstract constraints AC_1, AC_2 such that $c_1 \in \gamma(AC_1)$ and $c_2 \in \gamma(AC_2)$ it holds that $c_1 \wedge c_2 \in \gamma(AC_1 \wedge^{\mathcal{A}} AC_2)$.

With the concrete and abstract domains linked by a Galois connection or insertion, the safety condition can also be formulated in terms of the abstraction function:

—The abstract projection $\exists^{\mathcal{A}}_{-\tilde{x}}$ is a safe approximation of the concrete projection if for any set of constraints C and for any abstract constraint AC such that $\alpha(C) \leq^{\mathcal{A}} AC$ it holds that $\alpha(\exists_{-\tilde{x}} C) \leq^{\mathcal{A}} \exists^{\mathcal{A}}_{-\tilde{x}} AC$.

—The abstract conjunction $\wedge^{\mathcal{A}}$ is a safe approximation of the concrete conjunction if for any two sets of constraints C_1, C_2 and for any two abstract constraints AC_1, AC_2 such that $\alpha(C_1) \leq^{\mathcal{A}} AC_1$, $\alpha(C_2) \leq^{\mathcal{A}} AC_2$ it holds that $\alpha(C_1 \wedge C_2) \leq^{\mathcal{A}} AC_1 \wedge^{\mathcal{A}} AC_2$ where $C_1 \wedge C_2$ is the collecting conjunction, i.e., $C_1 \wedge C_2 = \{c_1 \wedge c_2 \mid c_1 \in C_1, c_2 \in C_2\}$.

Let a be a leaf atom of the AND-OR graph, and let AC be its abstract call constraint. Also, let $\rho_1, \ldots, \rho_m$ be the rules of the program P defining the predicate of a with the jth rule ρ_j of the form h_j:-$b_{j1}, \ldots, b_{jn_j}$. Basic transitions on the AND-OR graph are:

—*abstract_entry*(a, AC): This abstract transition has to approximate all entry transitions over LSLDT sequences $S :: \langle a; c\rangle$ with $c \in \gamma(AC)$. As explained in Section 4, for each rule ρ_j, the entry transition extends the sequence $S :: \langle a; c\rangle$ with the state $\langle b_{j1}, \ldots, b_{jn_j}; c^j_{in}\rangle$, where $c^j_{in} = \exists_{-vars(\rho_j)}(a = h_j \wedge c)$ (and creates an entry in Table with key $\langle a, c_{entry}\rangle$, where $c_{entry} = \exists_{-vars(a)} c$). Therefore, in this transition the leaf node a becomes an OR node, with the nodes h_j as children. A node h_j becomes an AND node with the atoms/constraints $b_{j1}, \ldots, b_{jn_j}$ as children. The abstract call constraints AC^j_{in} of b_{j1}, for all j, are computed. Finally, the transition computes AC_{entry}, an intermediate abstract constraint over $vars(a)$ approximating c_{entry}. The pair $\langle a, AC_{entry}\rangle$ will be a key in Table. This gives the following safety conditions:

—for AC_{entry}: $c \in \gamma(AC) \rightarrow c_{entry} \in \gamma(AC_{entry})$.

—for AC^j_{in}: $c \in \gamma(AC) \rightarrow c^j_{in} \in \gamma(AC^j_{in})$.

—*extension_from_table*$(a, AC, a^{tab}, AC^{tab})$: This abstract transition has to approximate all table lookup transitions on LSLDT sequences of the form $S_1 ::$

$\langle a, G; c\rangle$ for which there is an element in Table with key $\langle a^{tab}, c^{tab}_{entry}\rangle$ such that $\langle a^{tab}, c^{tab}_{entry}\rangle\delta = \langle a, c_{entry}\rangle$ for some renaming δ. For each stored answer c^{tab}_{exit}, LSLDT extends such a sequence with $\langle G; c \wedge c^{tab}_{exit}\delta\rangle$. The abstract transition has to compute the abstract success constraint AC' of a. With $(\langle a^{tab}, AC^{tab}_{entry}\rangle, AC^{tab}_{exit})$ the table entry such that $\langle a^{tab}, AC^{tab}_{entry}\rangle\delta = \langle a, AC_{entry}\rangle$ for some renaming δ, the safety condition is:

—$c \in \gamma(AC)$, $c^{tab}_{exit} \in \gamma(AC^{tab}_{exit})$, $(c^{tab}_{exit}\delta \to c_{entry}) \to c \wedge c^{tab}_{exit}\delta \in \gamma(AC')$.

—*abstract_exit*$(a, AC, \{h_1, \ldots, h_m\}, \{AC^1_{out}, \ldots, AC^m_{out}\})$: Let AC^j_{out} be the abstract out constraint of the rule ρ_j. This abstract transition has to approximate all exit transitions over LSLDT sequences of the form $S_1 :: \langle a, G; c\rangle \overset{\rho_j}{::} \langle b_{j1}, \ldots, b_{jn_j}; c^j_{in}\rangle :: S_2 :: \langle \Box_{\rho_j}; c^j_{out}\rangle$ where $c \in \gamma(AC)$, $c^j_{out} \in \gamma(AC^j_{out})$ and $c^j_{out} \to \exists_{-vars(\rho_j)}(c \wedge a = h_j)$. Such an exit transition computes $c_{exit} = \exists_{-vars(a)}(a = h_j \wedge c^j_{out})$ to be stored in Table as an answer for the key $\langle a, c_{entry}\rangle$ and extends the sequence with the state $\langle G; c \wedge c_{exit}\rangle$. The abstract transition has to compute AC_{exit}, the abstract constraint over $vars(a)$ to be stored as answer in Table for the element with key $\langle a, AC_{entry}\rangle$, and the abstract success constraint AC' of a. The safety conditions are:

—for AC_{exit}: $c \in \gamma(AC)$, $c^j_{out} \in \gamma(AC^j_{out})$, $(c^j_{out} \to \exists_{-vars(\rho_j)}(a = h_j \wedge c)) \to \exists_{-vars(a)}(a = h_j \wedge c^j_{out}) \in \gamma(AC_{exit})$.

—for AC': $c \in \gamma(AC)$, $c_{exit} \in \gamma(AC_{exit})$, $(c_{exit} \to c_{entry}) \to c \wedge c_{exit} \in \gamma(AC')$. Alternatively, the condition for AC' can be formulated without relying on AC_{exit}:
$c \in \gamma(AC)$, $c^j_{out} \in \gamma(AC^j_{out})$, $(c^j_{out} \to \exists_{-vars(\rho_j)}(a = h_j \wedge c)) \to \exists_{-vars(\rho_0)}(c \wedge a = h_j \wedge c^j_{out}) \in \gamma(AC')$.

A straightforward definition in terms of abstract projection, abstract conjunction, and constraint abstraction for the abstractions mentioned above, which satisfies the safety requirements, is:

—$AC_{entry} = \exists^{\mathcal{A}}_{-vars(a)} AC$,

—$AC^j_{in} = \exists^{\mathcal{A}}_{-vars(\rho_j)}(AC_{entry} \wedge^{\mathcal{A}} \alpha(a = h_j))$,

—$AC_{exit} = upp(AC^1_{exit}, \ldots, AC^m_{exit})$, where $AC^j_{exit} = \exists^{\mathcal{A}}_{-vars(a)}(AC^j_{out} \wedge^{\mathcal{A}} \alpha(a = h_j))$,

—$AC' = AC \wedge^{\mathcal{A}} AC_{exit}$, and in *extension_from_table* $AC' = AC \wedge^{\mathcal{A}} AC^{tab}\delta$ where δ is a renaming such that $a^{tab}\delta = a$. This computation is often called *extension*.

However, other definitions are feasible. As we will see later, different definitions can yield more accurate results, depending on the characteristics of the particular abstract domain considered.

Now we can describe the *call_to_success*(g, AC) procedure which controls a succession of transitions of which *abstract_entry* and *abstract_exit* are the most important ones. Assuming, without loss of generality, that a query consists of a single atom or constraint g with abstract call constraint AC, the abstract operational semantics (the AND-OR graph) is computed by *call_to_success*(g, AC).

If g is a constraint, then an AC' satisfying $c \in \gamma(AC) \rightarrow c \wedge g \in \gamma(AC')$ has to be computed. Defining AC' as $AC \wedge^{\mathcal{A}} \alpha(g)$ satisfies this condition. However, other definitions (e.g., not relying on α and $\wedge^{\mathcal{A}}$) are feasible.

If g is an atom, the procedure is as follows:

(1) Compute AC_{entry}.

(2) If Table has an entry $((\langle g^{tab}, AC^{tab}_{entry}\rangle, AC^{tab}_{exit})$ such that g^{tab} is a renaming of g and $AC^{tab}_{entry} \equiv^{\mathcal{A}} AC_{entry}$[7] (*table lookup*), then AC' is computed by *extension_from_table*$(g,\ AC,\ g^{tab},\ AC^{tab}_{exit})$.

(3) Else if there is an ancestor node g^{anc} with associated entry $(\langle g^{anc}, AC^{anc}_{entry}\rangle,$ $AC^{anc}_{exit})$ in Table such that g is a renaming of g^{anc} and for which *similar*$(AC^{anc}_{entry},$ $AC_{entry})$ holds (*table lookup*), then
—If $AC_{entry} \leq^{\mathcal{A}} AC^{anc}_{entry}$ then $AC' =$ *extension_from_table*$(g,\ AC,\ g^{anc},\ AC^{anc}_{exit})$.
—Else backtrack to g^{anc} and restart with *call_to_success*(g^{anc}, AC^{anc}) but with $AC^{anc}_{entry} = \mathcal{W}(AC^{anc}_{entry},\ AC_{entry})$.
(The original computation of *call_to_succes*$(g^{anc},\ AC^{anc})$ becomes obsolete.)

(4) Else
—Create an entry[8] $(\langle g, AC_{entry}\rangle, \perp)$ in Table.
—Apply *abstract_entry*(g, AC) obtaining the set of abstract *in constraints* $AC^1_{in}, \ldots, AC^m_{in}$, one for each rule $h_j \leftarrow B_j$ $(1 \leq j \leq m)$.
—The states $\langle B_j, AC^j_{in}\rangle$ are analyzed, applying, from left to right, *call_to_success* on the subgoals of the B_j. Eventually one obtains the abstract *out constraints* $AC^1_{out}, \ldots, AC^m_{out}$.
—Apply *abstract_exit*$(g, AC, \{h_1, \ldots, h_m\}, \{AC^1_{out}, \ldots, AC^m_{out}\})$. The intermediate result AC_{exit} is used to update the entry $(\langle g, AC_{entry}\rangle, AC^{tab}_{exit})$ of Table as follows.
If $AC_{exit} \leq^{A} AC^{tab}_{exit}$ then no update
Else if AC^{tab}_{exit} has already been used in a *table lookup* (this implies that g is a recursive predicate) then
—The new value is $\mathcal{W}(AC^{tab}_{exit}, AC_{exit})$.
—*Redo all computations* whose outcome depends directly or indirectly on the value AC^{tab}_{exit} which was used in the *table lookup*s (again part of the computations becomes obsolete). These are the "iterations" mentioned below. A crude way is to backtrack and to restart *call_to_success*(g, AC).
Else the new value is *upp*$(AC^{tab}_{exit}, AC_{exit})$.

The test *similar*$(AC^{anc}_{entry}, AC_{entry})$ must be such that no infinite chain of similar ancestors $\langle g, AC_{entry}\rangle, \langle g^{anc}, AC^{anc}_{entry}\rangle, \langle g^{anc^2}, AC^{anc^2}_{entry}\rangle, \ldots$ is created. A straightforward method is to put an arbitrary bound on the length of such chains. A more intelligent way would be to judge whether the differences among AC_{entry}, AC^{anc}_{entry}, and $\mathcal{W}(AC_{entry}, AC^{anc}_{entry})$ are significant with regard to the properties of interest (i.e., whether specialization for the different calls is worthwhile).

[7]Here and in the sequel we implicitly assume proper renaming of formulas.
[8]It can sometimes be preferable to enlarge AC_{entry}, for example because it contains uninteresting details or because there are already too many different entry patterns for g. If the enlarged $\langle g, AC^{enl}_{entry}\rangle$ can be solved by *table lookup*, then AC' is computed as in step (2).

The relevant information about the analysis (the atoms with their abstract entry and exit constraint) are all collected in Table. Abstract interpretation systems such as PLAI [Muthukumar and Hermenegildo 1990; 1992], GAIA [Englebert et al. 1992; Le Charlier et al. 1991] and AMAI [Janssens et al. 1995] do not construct the AND-OR graph explicitly. The systems use a more compact dependency structure which is sufficient to control the order of the transitions to be performed on the implicit AND-OR graph. Major difference among the systems is in the way they organize to "redo all computations dependent on an invalid *table lookup*": the way they attempt to minimize the number of transitions to be redone and how they attempt to make the best use of what has already been computed. Our PLAI implementation of the fixpoint algorithm [Hermenegildo et al. 1995; Muthukumar and Hermenegildo 1989; 1990; 1992] is performed as follows. The program is preprocessed in order to determine recursive predicates and recursive rules. This allows analyzing nonrecursive predicates in one pass without checking whether there is an ancestor node. For the recursive predicates, nonrecursive rules are analyzed first and once, and the result is taken as a first approximation of the answer. Then, the analysis for the recursive rules starts. The number of iterations performed in this computation is reduced by keeping track of the dependencies among nodes in the abstract AND-OR graph and the state of the information being computed. In some cases the fixpoint algorithm is able to finish in a single iteration.

5.3 Passive Constraints

The extended analysis framework proposed in the previous sections does not consider passive constraints. Integrating passive constraints in the concrete operational semantics can be done by using a more general representation of a state as a tuple $\langle G, c, s\rangle$ (s being a conjunction of constraints whose consistency has not been checked), modifying the conjunction operation so that it adds the constraints to s instead of to c, and including an $\mathit{infer}(c, s) = (c', s')$ step after each conjunction operation. This step moves active constraints from s to c and is immediately followed by a test for consistency [Jaffar and Maher 1994], at least if the considered CLP system is quick-checking.[9]

When considering the modifications needed at the abstract level, the fundamental question is what kind of information is required from the analysis and at what level of accuracy. Assume that gathering information regarding *which* constraints are passive and *when* they become active is not required from the analysis and that we prefer to lose accuracy rather than complicate the abstract operations. Then, the simplest method is to abstract both active and passive constraints by a single abstract component, without distinguishing between the information regarding passive constraints and that provided by the active constraints. This abstraction has to be safe with respect to all possible (future) activations of the passive constraint, and therefore it is possible to lose accuracy. However, this method significantly

[9]Special care is needed to perform safe analyses of systems that are not quick-checking. The problem that the analysis recognizes a state as a failure while the actual computation would proceed several steps, visiting several states that are not described by the output of the analysis, can be avoided by transforming the analyzed program so that it fails at run-time at the same point. Assuming absence of side effects, this is a transformation that cannot modify the observable behavior of the program and that always reduces run-time.

simplifies the abstract operations and allows such analysis to be integrated in the framework described above. We adopted this simple approach in the implementation of the analyzers presented in the following sections.

If the information provided by the analysis is aimed at detecting program points at which all constraints are definitely active, then we have to abstract in some way the *infer* function. In order to do this, the abstract domain should be able to approximate the information used by *infer* to decide if a constraint is definitely active. Then, for each constraint c analyzed, the abstract *infer* function must decide if under the current abstract constraint store, c is definitely active, and if it is not the case, it must abstract the fact that a passive constraint may appear (possibly without identifying which particular constraint it is) and the properties needed for such passive constraint to be definitely woken. Note that, if the domain is closed under antientailment, as is the case for definiteness analysis, then the approximation remains safe when a constraint is active before being recognized as such, so there is no need to deal with *possible* wake-ups of passive constraints. This option is closely related to the work in Hanus [1993] which presents an abstract domain for detecting CLP($\Re$) programs for which all passive nonlinear constraints eventually become linear at run-time. Otherwise, as for the freeness analysis, we must take possible wake-ups into account.

Finally, if the information is aimed at accurately modeling the delay and wake-up behavior, and we want to be able to determine which are the passive constraints, when they become passive, and when they are woken, we should split up the abstraction in two parts: an active part representing the active constraints and a passive part representing the passive constraints. In this case, the abstract projection function has to preserve enough information to ensure the correct wake-up behavior. A possible technique is to project only the abstract active constraints and to keep the passive part. Then an abstract constraint is no longer restricted to a finite number of variables (the variables of the rule, goal, or query) as it is in the original abstract interpretation framework. As a consequence, termination is not guaranteed, and some new kind of widening should be introduced. This is related to the work of Marriott et al. [1994], which gives a simple denotational semantics and a generic global data-flow analysis algorithm which is based on the semantics sketched above, for languages in which the computation generally proceeds left to right but in which some calls are dynamically delayed until their arguments are sufficiently instantiated, a very similar case to that of the passive constraints. An alternative technique which is able to project both active and passive components while maintaining accuracy has been recently described in García de la Banda et al. [1995].

6. INFERENCE OF DEFINITENESS INFORMATION

In this section we present the abstract domain $Cons^{\mathcal{D}}$, which approximates definiteness information in CLP programs, and the corresponding abstract functions as required for the extended framework developed above. The abstraction is based on a high-level description of definiteness dependencies which are easy to obtain for each particular type of constraint in an actual system. We have attempted to give intuitively comprehensible definitions of the different operations, rather than algorithmic versions. The algorithms can be found in García de la Banda [1994], where proofs of correctness for such algorithms are also provided.

6.1 Abstract Domain and Abstraction Function

Let $\wp(S)$ denote the powerset of a set S, and let $\wp_\emptyset(S)$ denote $\wp(S) \setminus \{\emptyset\}$. Also, let Var denote a denumerable set of variables and $Pvar \subseteq Var$ a distinguished (denumerable) set of variables which may occur in programs. An abstract constraint $AC^{\mathcal{D}} = (D, R)$ of the abstract domain $Cons^{\mathcal{D}}$ is an element of $\wp(Pvar) \times \wp(Pvar \times \wp_\emptyset(\wp_\emptyset(Pvar)))$ which is in simplified form. A variable x in D represents a variable that is known to be definite, which can be represented by the propositional formula $x \leftarrow true$. An element $(x, \{S_1, \ldots, S_n\}) \in R$ with $S_i = \{x_{i1}, \ldots, x_{im_i}\}$ represents known dependencies between variables. These dependencies can be expressed by the propositional formula $x \leftarrow conj(S_1) \vee \ldots \vee conj(S_n)$ where $conj(S_i) = x_{i1} \wedge \ldots \wedge x_{im_i}$ (this is equivalent with $(x \leftarrow conj(S_1)) \wedge \ldots \wedge (x \leftarrow conj(S_n))$), where a formula $x \leftarrow conj(S_i)$ expresses that x is definite if x_{i1} up to x_{im_i} are. An element (D, R) is in simplified form if it encodes at most one formula $x \leftarrow \ldots$ for each variable x and has an explicit representation of all implied nonredundant formulas of the form $x \leftarrow conj(S)$. A formula $x \leftarrow conj(S)$ is considered redundant if it is a tautology (i.e., $x \in S$) or if it is implied by another formula $x \leftarrow conj(S')$ (i.e., $S' \subseteq S$). Putting formulas in simplified form gives a more compact representation and reduces the cost of key operations, such as testing for equivalence and performing abstract projection. A simplified form can be obtained by applying the following rewrite rules:

(1) $(D, \{(x, SS_1)\} \cup \{(x, SS_2)\} \cup R) \Rightarrow (D, \{(x, SS_1 \cup SS_2)\} \cup R)$.
(2) $(D, \{(x, \{S_1\} \cup \{S_2\} \cup SS)\} \cup R) \Rightarrow (D, \{(x, \{S_1\} \cup SS)\} \cup R)$ if $S_1 \subset S_2$.
(3) $(D, \{(x, SS)\} \cup R) \Rightarrow (D, R)$ if $x \in D$.
(4) $(D, \{(x, \{\{y\} \cup S\} \cup SS)\} \cup R) \Rightarrow (D, \{(x, \{S\} \cup SS)\} \cup R)$ if $y \in D$.
(5) $(D, \{(x, \{\emptyset\} \cup SS)\} \cup R) \Rightarrow (\{x\} \cup D, R)$.
(6) $(D, \{(x, \{\{y\} \cup S_1\} \cup SS_1)\} \cup \{(y, \{S_2\} \cup SS_2)\} \cup R) \Rightarrow (D, \{(x, \{S_1 \cup S_2\} \cup \{\{y\} \cup S_1\} \cup SS_1)\} \cup \{(y, \{S_2\} \cup SS_2)\} \cup R)$ if $x \notin S_2$ and $\nexists S \in SS_1$ such that $S \subseteq (S_1 \cup S_2)$.

Rule (1) merges several definite dependencies approximated for the same variable. Knowing that the definiteness of x can be derived from the definiteness of a set S_2 of variables is useless once the definiteness of x is known to be derived from a subset S_1 of S_2. Rule (2) eliminates those useless S_2 sets. Approximating that the definiteness of x can be derived from the definiteness of any other set of variables is useless once x is in D. Rule (3) performs such simplification. If a variable y in a set $S \in SS$ of (x, SS) is in D, y can be removed from S without losing information. Rule (4) removes those variables. The element $(x, \{\emptyset\} \cup SS)$ is obtained once x is known to be definite. Rule (5) eliminates $(x, \{\emptyset\} \cup SS)$ from R and adds x to D. If the definiteness of y can be inferred from that of the variables in S_2, and the definiteness of x can in turn be inferred from that of $\{y\} \cup S_1$, we can conclude that the definiteness of x can also be inferred from that of $S_2 \cup S_1$. This propagation of definiteness dependencies is performed by rule (6). Note that the condition "$\nexists S \in SS_1$ such that $S \subseteq (S_1 \cup S_2)$" avoids infinite applications of rule (6) (if $S = S_1 \cup S_2$) or infinite alternate applications of rules (6) and (2) (if $S \subset S_1 \cup S_2$).

Let $simplify(D,R)$ denote the abstract constraint obtained by applying the rewrite rules to (D,R) until no rule can be applied. We can now formally define $Cons^{\mathcal{D}}$ as $\{\bot\} \cup \{(D,R) \in \wp(Pvar) \times \wp(Pvar \times \wp_{\emptyset}(\wp_{\emptyset}(Pvar))) \mid simplify(D,R) = (D,R)\}$.[10] For convenience, in the rest of the section we will denote by $min\mathcal{D}(SS)$ the set of sets obtained by applying rule (2) to a particular SS of $(x,SS) \in R$.

Definition 6.1.1 (*Abstraction of a Constraint:* α^d). Let c be a constraint. Then $\alpha^d(c) = \bot$ if $\neg consistent(c)$; otherwise $\alpha^d(c) = (D,R)$ where

(1) $D = def(c)$[11]

(2) $R = \{(x,SS) \mid x \in vars(c), SS = min\mathcal{D}(gr\_dep(c,x)), SS \neq \emptyset, SS \neq \{\emptyset\}\}$

(3) $gr\_dep(c,x) = \{\tilde{y} \subseteq vars(c) \setminus \{x\} \mid$ *for all sequences of values* $\tilde{v}$ *s.t.* $consistent(c \wedge \tilde{y} = \tilde{v})$, *holds that* $x \in def(c \wedge \tilde{y} = \tilde{v})\}$.

Note that $\emptyset \in gr\_dep(c,x)$ for any $x \in def(c)$ and for any x such that no definite dependency can be found. In such cases $min\mathcal{D}(gr\_dep(c,x)) = \{\emptyset\}$.

Example 6.1.2. Note that the symbol "." stands for concatenation of PrologIII lists and that "$< y >$" is a list with one element.

$$
\begin{array}{lcl}
\alpha^d(x = 3) & = & (\{x\}, \emptyset) \\
\alpha^d(x = f(y,z)) & = & (\emptyset, \{(x,\{\{y,z\}\}), (y,\{\{x\}\}), (z,\{\{x\}\})\}) \\
\alpha^d(x = 3y + 2z) & = & (\emptyset, \{(x,\{\{y,z\}\}), (y,\{\{x,z\}\}), (z,\{\{x,y\}\})\}) \\
\alpha^d(x > y) & = & (\emptyset, \emptyset) \\
\alpha^d(x \neq y) & = & (\emptyset, \emptyset) \\
\alpha^d(x = y * z) & = & (\emptyset, \{(x,\{\{y,z\}\})\}) \\
\alpha^d(x = < y > .z) & = & (\emptyset, \{(x,\{\{y,z\}\}), (y,\{\{x\}\}), (z,\{\{x\}\})\}) \\
\alpha^d(x = < y > . < w > .z) & = & (\emptyset, \{(x,\{\{y,w,z\}\}), (y,\{\{x\}\}), \\
 & & (w,\{\{x\}\}), (z,\{\{x\}\})\}) \\
\alpha^d(x = < y > .w.z) & = & (\emptyset, \{(x,\{\{y,w,z\}\}), (y,\{\{x\}\}), \\
 & & (w,\{\{x,z\}\}), (z,\{\{x,w\}\})\})
\end{array}
$$

Definition 6.1.3 (*Order Relation*). Let $(D_1,R_1),(D_2,R_2) \in Cons^{\mathcal{D}}$. Then $(D_1,R_1) \leq^{\mathcal{D}} (D_2,R_2)$ iff:

(1) $D_2 \subseteq D_1$

(2) $\forall (x,SS_2) \in R_2 : x \in D_1$ or $(\exists (x,SS_1) \in R_1$ such that $\forall S_2 \in SS_2 : \exists S_1 \in SS_1,\ S_1 \subseteq S_2)$.

Intuitively, this means that for every formula represented by (D_2,R_2), there is a formula in (D_1,R_1) which is at least as strong.

Definition 6.1.4 (*Equivalence*). Let $(D_1,R_1),(D_2,R_2) \in Cons^{\mathcal{D}}$. Then $(D_1,R_1) \equiv^{\mathcal{D}} (D_2,R_2)$ iff:

(1) $D_1 = D_2$

(2) $R_1 = R_2$.

---

[10] For reasons of readability most of the following definitions and operations do not explicitly deal with $\bot$. Their extensions are trivial.

[11] As mentioned in Section 2.1, $def(c)$ denotes the set of definite variables in c.

Definition 6.1.5 (*Least Upper Bound*). Let $(D_1, R_1), (D_2, R_2) \in Cons^{\mathcal{D}}$. Then $upp^{\mathcal{D}}((D_1, R_1), (D_2, R_2)) = (D, R)$ where

(1) $D = D_1 \cap D_2$

(2) $R = \{(x, SS) \in R_i \mid x \in D_j, i, j \in \{1, 2\}, i \neq j\} \cup \{(x, min\mathcal{D}(SS')) | \; SS' = \{S_1 \cup S_2 \mid (x, SS_1) \in R_1, S_1 \in SS_1, \, (x, SS_2) \in R_2, S_2 \in SS_2\}\}$.

The definition can easily be extended to compute the least upper bound of m ($m > 2$) abstractions. In the following we will assume that the function *upp* applies to a set of abstract constraints.

Definition 6.1.6 (*Abstraction of a Set of Constraints:* $\alpha^{\mathcal{D}}$). Let $C \in Cons^{\mathcal{C}}$. Then $\alpha^{\mathcal{D}}(C) = \bot$ if $C = \emptyset$; otherwise $\alpha^{\mathcal{D}}(C) = upp(\{\alpha^{d}(c) \mid c \in C\})$.

Definition 6.1.7 (*Maximal and Minimal Elements*). The maximal element is $\top = (\emptyset, \emptyset)$. The minimal element is $\bot$, denoting the empty set of constraints.

The concretization function $\gamma^{\mathcal{D}}$ can be defined based upon $\alpha^{\mathcal{D}}$ as described in Cousot and Cousot [1992a]: $\gamma^{\mathcal{D}}(AC) = \bigcup\{C \in Cons^{\mathcal{C}} \mid \alpha^{\mathcal{D}}(C) \leq^{\mathcal{D}} AC\}$. Then $(Cons^{\mathcal{C}}, \subseteq, Cons^{\mathcal{D}}, \leq^{\mathcal{D}})$ is a Galois insertion [García de la Banda and Hermenegildo 1993].

6.2 Abstract Projection and Abstract Conjunction Functions

Definition 6.2.1 (*Abstract Projection*). Let $(D_1, R_1) \in Cons^{\mathcal{D}}$ and $\tilde{x}$ be a set of variables. Then $\exists^{\mathcal{D}}_{-\tilde{x}}(D_1, R_1) = (D, R)$ where

(1) $D = D_1 \cap \tilde{x}$

(2) $R = \{(x, SS) \mid (x, SS_1) \in R_1, x \in \tilde{x}, SS = \{S \in SS_1 \mid S \subseteq \tilde{x}\}, SS \neq \emptyset\}$.

The propositional formula represented by (D, R) is the projection of the formula represented by (D_1, R_1). Intuitively, D is the subset of variables in $\tilde{x}$ which are known to be definite in D_1, and R contains the definiteness dependencies (if any) approximated by R_1 for the possibly nondefinite variables in $\tilde{x}$. Since only the variables in $\tilde{x}$ are taken into account, any element $(y, SS_1) \in R_1$ approximating the dependencies for a variable which is not in $\tilde{x}$ (i.e., $y \notin \tilde{x}$) is eliminated. Furthermore, the dependency sets in SS_1 of the elements $(x, SS_1) \in R_1, x \in \tilde{x}$ which are not subsets of $\tilde{x}$ are also eliminated, as groundness of *all* variables in a dependency set is required to ground x, yielding SS. Note that if as a result SS becomes empty, there is no information for the definiteness dependencies of x w.r.t. the variables in $\tilde{x}$, and no (x, SS) will appear in R.

Definition 6.2.2 (*Abstract Conjunction*). Let $(D_1, R_1), (D_2, R_2) \in Cons^{\mathcal{D}}$. Then $(D_1, R_1) \wedge^{\mathcal{D}} (D_2, R_2) = simplify(D_1 \cup D_2, R_1 \cup R_2)$.

A more implementation oriented definition of the abstract conjunction function would state that we should first apply rule (1), then rules (3), (4), and (5) (thus propagating definiteness), and finally rule (6) (propagating definite dependencies). Note that we may also need to apply rule (2) immediately after the application of rules (1), (4), or (6). The order in which those steps are performed has been chosen to increase efficiency, but they can be performed in any order affecting neither correctness nor accuracy. For a more implementation oriented definition of this operation, see García de la Banda [1994].

Example 6.2.3. Consider the abstract constraints:

(D_1, R_1)	$(\emptyset, \{(x, \{\{y\}, \{z\}\}), (y, \{\{z\}\})\})$
(D_2, R_2)	$(\{z\}, \{(y, \{\{w\}\}), (w, \{\{y\}\})\})$

Then $(D_1, R_1) \wedge^{\mathcal{D}} (D_2, R_2)$ yields the abstract constraint (D, R) as follows:

$$\begin{array}{l}
\mathit{simplify}(\{z\}, \{(x, \{\{y\}, \{z\}\}), (y, \{\{z\}\}), (y, \{\{w\}\}), (w, \{\{y\}\})\}) \rightarrow^1 \\
\mathit{simplify}(\{z\}, \{(x, \{\{y\}, \{z\}\}), (y, \{\{z\}, \{w\}\}), (w, \{\{y\}\})\}) \rightarrow^4 \\
\mathit{simplify}(\{z\}, \{(x, \{\{y\}, \emptyset\}), (y, \{\{z\}, \{w\}\}), (w, \{\{y\}\})\}) \rightarrow^4 \\
\mathit{simplify}(\{z\}, \{(x, \{\{y\}, \emptyset\}), (y, \{\emptyset, \{w\}\}), (w, \{\{y\}\})\}) \rightarrow^5 \\
\mathit{simplify}(\{x, z\}, \{(y, \{\emptyset, \{w\}\}), (w, \{\{y\}\})\}) \rightarrow^5 \\
\mathit{simplify}(\{x, y, z\}, \{(w, \{\{y\}\})\}) \rightarrow^4 \\
\mathit{simplify}(\{x, y, z\}, \{(w, \{\emptyset\})\}) \rightarrow^5 \\
\mathit{simplify}(\{x, y, z, w\}, \emptyset) = (\{x, y, z, w\}, \emptyset)
\end{array}$$

where $\rightarrow^n$ represents the application of the nth rule. Thus $(D_1, R_1) \wedge^{\mathcal{D}} (D_2, R_2) = (\{x, y, z, w\}, \emptyset)$

Consider now the abstract constraints:

(D_1, R_1)	$(\emptyset, \{(x, \{\{y\}, \{z, w\}\}), (y, \{\{z, w\}\})\})$
(D_2, R_2)	$(\emptyset, \{(y, \{\{z\}\}), (z, \{\{y\}\})\})$.

Then $(D_1, R_1) \wedge^{\mathcal{D}} (D_2, R_2)$ yields the abstract constraint (D, R) as follows:

$$\begin{array}{l}
\mathit{simplify}(\emptyset, \{(x, \{\{y\}, \{z, w\}\}), (y, \{\{z, w\}\}), \{(y, \{\{z\}\}), (z, \{\{y\}\})\}) \rightarrow^1 \\
\mathit{simplify}(\emptyset, \{(x, \{\{y\}, \{z, w\}\}), (y, \{\{z, w\}, \{z\}\}), (z, \{\{y\}\})\}) \rightarrow^2 \\
\mathit{simplify}(\emptyset, \{(x, \{\{y\}, \{z, w\}\}), (y, \{\{z\}\}), (z, \{\{y\}\})\}) \rightarrow^6 \\
\mathit{simplify}(\emptyset, \{(x, \{\{y\}, \{z\}, \{z, w\}\}), (y, \{\{z\}\}), (z, \{\{y\}\})\}) \rightarrow^2 \\
\mathit{simplify}(\emptyset, \{(x, \{\{y\}, \{z\}\}), (y, \{\{z\}\}), (z, \{\{y\}\})\}) = \\
(\emptyset, \{(x, \{\{y\}, \{z\}\}), (y, \{\{z\}\}), (z, \{\{y\}\})\})
\end{array}$$

Thus $(D_1, R_1) \wedge^{\mathcal{D}} (D_2, R_2) = (\emptyset, \{(x, \{\{y\}, \{z\}\}), (y, \{\{z\}\}), (z, \{\{y\}\})\})$.

Let us now present how the abstractions required by the framework are computed. Let g be a constraint or an atom and AC be its abstract call constraint. If g is a constraint, then AC' is defined as $AC \wedge^{\mathcal{D}} \alpha^d(g)$. If g is an atom, let $\rho_1, \ldots, \rho_m$ be the rules of the program P defining the predicate of g, ρ_j be h_j:-$b_{j1}, \ldots, b_{jn_j}$, and let $AC^1_{out}, \ldots, AC^m_{out}$ be the abstract out constraints of rules $\rho_1, \ldots, \rho_m$ respectively. Then, the abstract entry, in, exit, and success constraints are defined as follows:

—$AC_{entry} = \exists^{\mathcal{D}}_{-vars(g)} AC$,

—$AC^j_{in} = \exists^{\mathcal{D}}_{-vars(\rho_j)} (AC_{entry} \wedge^{\mathcal{D}} \alpha^d(g = h_j))$,

—$AC_{exit} = upp^{\mathcal{D}}(AC^1_{exit}, \ldots, AC^m_{exit})$, where $AC^j_{exit} = \exists^{\mathcal{D}}_{-vars(g)} (AC^j_{out} \wedge^{\mathcal{D}} AC_{entry} \wedge^{\mathcal{D}} \alpha^d(g = h_j))$,

—$AC' = AC \wedge^{\mathcal{D}} AC_{exit}$, and in *extension_from_table* $AC' = AC \wedge^{\mathcal{D}} AC^{tab} \delta_{a = a^{tab}}$.

It is clear that all definitions satisfy the safety requirements imposed by the framework. However, two important issues must be pointed out. The first issue is related to one of the three properties of the abstract operations identified in Jacobs and Langen [1992]: *additivity*. This property requires that precision should not be lost when commuting the least upper bound with an abstract operation. Additive upper bounds are not common, and $upp^{\mathcal{D}}$ is not an exception. As a result, it is possible to obtain a more accurate AC' by computing AC' as $upp(AC \wedge^{\mathcal{D}} AC^1_{exit}, \ldots, AC \wedge^{\mathcal{D}} AC^m_{exit})$. However, the price is m applications of $\wedge^{\mathcal{D}}$ instead of one. As for this analysis abstract conjunction is an expensive computation, this approach is not taken.

The second issue is related to the definition of AC_{exit} and, in particular, to the appearance of AC_{entry} in the definition of each AC^j_{exit}.[12] This redundant constraint is added in order to avoid a loss of precision caused by the interaction among approximating a property that is closed under antientailment (downward-closed), nonnormalization, a loss of precision in the abstract projection function, and the tabulation method. Let us illustrate the problem with a simple example.

Example 6.2.4. Assume we have a program P with only one rule $\rho_1 : p(z)$. (i.e., a fact). The computation of $call\_to\_success(p(f(x,y)), AC)$, where $AC = (\{x\}, \emptyset)$, will proceed as follows:

(1) $abstract\_entry(p(f(x,y)), AC)$. Following the definitions above, we will obtain $AC_{entry} = (\{x\}, \emptyset)$ and $AC^1_{in} = (\emptyset, \emptyset)$.
(2) Since the body of ρ_1 is empty, $AC^1_{out} = AC^1_{in} = (\emptyset, \emptyset)$.
(3) $abstract\_exit(p(f(x,y)), AC, \{p(z)\}, \{AC^1_{out}\})$. If we compute AC^1_{exit} as $\exists^{\mathcal{D}}_{-vars(a)} (AC^j_{out} \wedge^{\mathcal{D}} \alpha^d(f(x,y) = p(z)))$, we will obtain $AC^1_{exit} = (\emptyset, \emptyset)$. Then, $AC_{exit} = (\emptyset, \emptyset)$ and $AC' = (\{x\}, \emptyset)$. On the other hand, if we include AC_{entry} in the definition of AC^1_{exit} (as proposed in the above definitions), we obtain $AC_{exit} = AC^1_{exit} = (\{x\}, \emptyset)$, thus avoiding a loss of precision.

Although accuracy is always recovered when computing AC', the difference can have an adverse effect on memory (tabulation) and time consumption (computing AC'). Also, for some applications it is convenient that AC_{exit} provides accurate information about the success state (for example, the output mode of the predicate). Finally, the loss of accuracy in AC_{exit} could imply a greater number of fixpoint iterations, since they depend on the value of AC_{exit}. Regarding the extra cost introduced by our definition, note that since $AC_{entry} \wedge^{\mathcal{A}} \alpha^d(g = h_j)$ is already computed during the *abstract_entry* operation, the alternative computation does not introduce a significant overhead.

As a last remark, we use $upp^{\mathcal{D}}$ as a widening operator, since $Cons^{\mathcal{D}}$ (when considered over a finite set of variables) does not have infinite ascending chains.

There are at least two other domains which are closely related to ours. One is the domain proposed by Hanus [1995] and originally used for detecting situations in which the residuation rule[13] can be guaranteed to never be activated in a given

[12] Recall that AC^j_{exit} can be defined in a simpler way, such as $\exists^{\mathcal{D}}_{-vars(g)}(AC^j_{out} \wedge^{\mathcal{D}} \alpha^d(g = h_j))$, while still satisfying the safety conditions.
[13] Residuation is an operational mechanism for the integration of functions into logic programming.

program (this is similar in some ways to a "nonsuspension" analysis, as the residuation rule delays the evaluation of functions during the unification process until the arguments are sufficiently instantiated). The nonresiduation requirements imply groundness requirements for the arguments of certain functions, and a domain similar to the one defined in this section is used for inferring such groundness.

The second related domain is the domain of positive Boolean functions which are closed under intersection. This domain was defined early on by Dart [1988] under the name of *dependency formulae* and applied to the inference of groundness in deductive databases. Our domain can be seen as a compact representation of this domain (including a formulation of efficient operations for it). The main difference is that, for efficiency reasons, we require the abstraction to be in a particular simplified form. Recently, the different possible subsets of the Boolean functions which can be used for tracking dependencies in program analysis and their representations have been studied and greatly clarified [Armstrong et al. 1994]. Our domain corresponds essentially to the *Def* domain in this taxonomy. The work developed in Armstrong et al. [1994] also illustrates that the representation that we have proposed is closely related to the *CDF* representation which is shown therein to offer an advantageous cost-performance tradeoff.

7. INFERENCE OF FREENESS INFORMATION

The definiteness analysis infers whether variables are *definite*, i.e., constrained to a unique value. The analysis takes into account *definite* dependencies among variables in order to perform accurate definiteness propagation. The freeness analysis derives whether variables are *free*, i.e., whether they can range over the whole domain specified by their type: e.g., a variable that is constrained to be numerical but still ranges over the complete domain of numbers is considered as free. It keeps track of *possible* dependencies between variables to take care of nonfreeness propagation: in order to obtain definite freeness information we must trace all possible dependencies. These dependencies are established via the constraints in the program either directly or through entailment. The derived information is useful for example to perform constraint reordering (see Dumortier [1994]).

The most closely related work to our freeness analysis is the *LSign* abstraction of Marriott and Stuckey [1993] that describes sets of linear equations and inequalities. In Marriott and Stuckey [1994], this domain is further elaborated and extended toward the treatment of nonlinear constraints and unification constraints. The major advantage of the abstraction compared with ours is its enhanced precision, especially for inequalities but also for equations (it keeps track of the constraint symbol and the sign of the coefficients, which are discarded in our analysis). However, the main deficiencies are that (1) no implementation is reported, such that the efficiency (especially with respect to the increased precision) cannot be judged and (2) some aspects that are relevant in order to obtain a complete analyzer are not (sufficiently) elaborated (such as procedure-exit, the upper bound operation, the order relation, and the interaction between the unification and the numerical part). Recently, Ramachandran and Van Hentenryck [1995] described some improvements.

7.1 Abstract Domain and Abstraction Function

Let c denote a constraint. A set of variables $\{x_1, \ldots, x_n\} \subseteq vars(c)$ is *constrained* by c iff there exists a set of values $\{v_1, \ldots, v_n\}$, with each v_i in the domain of x_i, such that $c \wedge x_1 = v_1 \wedge \ldots \wedge x_n = v_n$ is inconsistent while for any $\{i_1, \ldots, i_m\} \subset \{1, \ldots, n\}$ it holds that $c \wedge x_{i_1} = v_{i_1} \wedge \ldots \wedge x_{i_m} = v_{i_m}$ is consistent.

Example 7.1.1. Let c be $x = f(y_1, \ldots, y_n)$ $(n \geq 0)$. The sets constrained by c are $\{x\}$, $\{x, y_1\}, \ldots, \{x, y_n\}$ (e.g., for $\{x, y_1\}$, $c \wedge x = f(1, \ldots, n) \wedge y_1 = 3$ is inconsistent while any subpart of the conjunction is consistent). Let c be $a_1x_1 + \ldots + a_nx_n = b$ $(n \geq 1)$ where the a_i and b are numbers $(a_i \neq 0)$. Then c constrains the set $\{x_1, \ldots, x_n\}$. Let c be $x > y$. Then c constrains $\{x, y\}$. Let c be $x = y * z$. Then c constrains $\{x, y\}$, $\{x, z\}$, and $\{x, y, z\}$ (for $\{x, z\}$, $c \wedge x = 1 \wedge z = 0$ is inconsistent while $c \wedge x = 1$ and $c \wedge z = 0$ are consistent; for $\{x, y, z\}$, $c \wedge x = 2 \wedge y = 1 \wedge z = 1$ is inconsistent while any subpart of the conjunction is consistent). Let c be $x =< y > .w.z$. The sets constrained by c are $\{x, y\}, \{x, w\}$, and $\{x, z\}$.

A variable x is *free* in c iff $\{x\}$ is not constrained by c, so freeness can be derived by safely approximating all possible constrained sets. A constrained set $\{x_1, \ldots, x_n\}$ with $n > 1$ indicates a possible dependency between those variables in the sense that constraining all variables, but for example x_i, can constrain x_i (can cause nonfreeness of x_i). Such constrained sets are the key concept used to perform nonfreeness propagation. The formal development in Dumortier [1994] (which is too long to include) shows that constrained sets that can be obtained as union of others (e.g., the set $\{x, y, z\}$ in the last example), and unions of constrained sets, are redundant with respect to nonfreeness propagation (the subdependencies impose stronger restrictions). These *nonminimal* sets can therefore be omitted in the abstraction.[14]

Definition 7.1.2 (*Minimal Set*). Let $SS \in \wp(\wp_\emptyset(Pvar))$. Then $S \in SS$ is minimal in SS iff $\nexists S_1, \ldots, S_m \in SS \setminus \{S\}$ $(m \geq 2)$ such that $S = S_1 \cup \ldots \cup S_m$.

Definition 7.1.3 ($min\mathcal{F}$). Let $SS \in \wp(\wp_\emptyset(Pvar))$. Then $min\mathcal{F}(SS) = \{S \in SS \mid S \textit{ is a minimal set in } SS\}$.

Definition 7.1.4 (*Abstraction of a Constraint:* α^f). Let c be a constraint. Then $\alpha^f(c) = \bot$ if $\neg consistent(c)$; otherwise $\alpha^f(c) = min\mathcal{F}(\{\{x_1, \ldots, x_n\} \subseteq vars(c) \mid \{x_1, \ldots, x_n\} \textit{ is constrained by } c\})$.

The abstract domain $Cons^{\mathcal{F}^m}$ can now be formally defined as $\{\bot\} \cup \{AC \in \wp(\wp_\emptyset(Pvar)) \mid min\mathcal{F}(AC) = AC\}$.[15]

[14] The nonminimal freeness abstraction of a constraint c as developed in Dumortier et al. [1993] exhaustively enumerates not only minimal constrained sets in c but also all possible unions of these. These unions are needed at abstract conjunction (see Definition 7.2.2). Adding the unions at once instead of computing them at abstract conjunction contributes to the precision of the analysis. However, it also limits its practical use, as the size of the abstractions is in the worst case exponential with respect to the number of variables.

[15] For reasons of readability most of the following definitions and operations do not explicitly deal with $\bot$. Their extensions are trivial.

While it is rather straightforward to derive the abstraction of *primitive* constraints, it is more involved for a *conjunction* of primitive constraints. Let us consider some examples.

Example 7.1.5. Let c be $y = f(g(x)) \wedge z = x$. Constrained sets are $\{y\}$ and $\{y, x\}$ (from the first primitive constraint) and $\{z, x\}$ (from the second primitive constraint), but also $\{y, z\}$ from the entailed primitive constraint $y = f(g(z))$. Let c be $x + y = 3 \wedge y - z = 2$. Constrained sets are $\{x, y\}$ and $\{y, z\}$ but also $\{x, z\}$, as there is an entailed primitive constraint $x + z = 1$.

This suggests that it is sufficient to consider the constrained sets for all entailed primitive constraints. However, this does not suffice for conjunctions composed of constraints of different constraint domains, as shown by the following example.

Example 7.1.6. Let c be $x = f(u, v) \wedge u - v + t = 3$. Besides the constrained sets of the first conjunct ($\{x\}$, $\{x, u\}$, and $\{x, v\}$) and of the second conjunct ($\{u, v, t\}$), there is also a constrained set $\{x, t\}$. Indeed, for example, $c \wedge x = f(1, 2) \wedge t = 1$ is inconsistent while any subpart of the conjunction is consistent.

In our implementation, we have not attempted to compute constrained sets of nonprimitive constraints, but rather use abstract conjunction to obtain their abstraction from the abstractions of the composing conjuncts. It is recommended to first put the conjunction in solved form, as the presence of redundant conjuncts will severely affect precision.[16] Even in the absence of redundancy, one can obtain a more precise result when starting from the solved form, as will be illustrated below. For the Herbrand domain, the solved form can be obtained by applying the Martelli-Montanari unification algorithm [Martelli and Montanari 1982]; for generalized linear constraints, a solved form can be obtained by the algorithm of Lassez and McAloon [1992].

Before discussing abstract conjunction, let us first further develop the abstract domain.

Definition 7.1.7 (*Order Relation*). Let AC_1, $AC_2 \in Cons^{\mathcal{F}^m}$. Then $AC_1 \leq^{\mathcal{F}^m} AC_2$ iff $AC_1 \subseteq close(AC_2)$ where $close(AC)$ is the closure under union of AC.

Definition 7.1.8 (*Equivalence*). Let AC_1, $AC_2 \in Cons^{\mathcal{F}^m}$. Then $AC_1 \equiv^{\mathcal{F}^m} AC_2$ iff $AC_1 = AC_2$.

Definition 7.1.9 (*Least Upper Bound*). Let AC_1, $AC_2 \in Cons^{\mathcal{F}^m}$. Then $upp^{\mathcal{F}^m}(AC_1, AC_2) = min\mathcal{F}(AC_1 \cup AC_2)$.

This definition can easily be extended to compute the least upper bound of m ($m > 2$) abstractions. In the following we will assume that upp applies to a set of abstract constraints.

[16] This is not done in the actual implementation based on the PLAI system, which is written in Prolog. In this case the only highly efficient solved-form algorithm readily available in the system itself is the one for unification constraints inherited from the Prolog implementation. However, as pointed out in Codognet and Filé [1992], implementing the system in the CLP language to be analyzed would allow to use all built-in solved-form algorithms. On the other hand it should also be noted that for the actual benchmarks analyzed in Section 9 not applying the solved-form algorithm does not affect precision.

To abstract a set of constraints, ideally $\wp(\wp(\wp_\emptyset(Pvar)))$ should be the abstract domain. However, this may give rise to impractically large abstractions. Therefore, the abstraction of a set of constraints is approximated by the least upper bound of the abstractions of the individual constraints in the set.

Definition 7.1.10 (*Abstraction of a Set of Constraints:* $\alpha^{\mathcal{F}^m}$). Let $C \in Cons^{\mathcal{C}}$. Then $\alpha^{\mathcal{F}^m}(C) = \bot$ if $C = \emptyset$; otherwise $\alpha^{\mathcal{F}^m}(C) = upp(\{\alpha^f(c) \mid c \in C\})$.

Definition 7.1.11 (*Maximal and Minimal Elements*). The maximal element is $min\mathcal{F}(\wp(\wp_\emptyset(Pvar))) = \{\{x\} \mid x \in Pvar\}$. The minimal element is $\bot$.

The concretization function $\gamma^{\mathcal{F}^m}$ can be defined based upon $\alpha^{\mathcal{F}^m}$ as described in Cousot and Cousot [1992a]: $\gamma^{\mathcal{F}^m}(AC) = \bigcup\{C \in Cons^{\mathcal{C}} \mid \alpha^{\mathcal{F}^m}(C) \leq^{\mathcal{F}^m} AC\}$. Then $(Cons^{\mathcal{C}}, \subseteq, Cons^{\mathcal{F}^m}, \leq^{\mathcal{F}^m})$ is a Galois insertion [Dumortier 1994].

7.2 Abstract Projection and Abstract Conjunction Functions

Definition 7.2.1 (*Abstract Projection*). Let $AC \in Cons^{\mathcal{F}^m}$ and $\tilde{x}$ be a sequence of variables. Then $\exists^{\mathcal{F}^m}_{-\tilde{x}} AC = \{S \in AC \mid S \subseteq \tilde{x}\}$.

The abstract conjunction of two abstract constraints AC_1 and AC_2, denoted $AC_1 \wedge^{\mathcal{F}^m} AC_2$, must safely approximate the constrained sets of all constraints $c_1 \wedge c_2$ where c_1 and c_2 are abstracted by AC_1 and AC_2 respectively. It is obvious that constrained sets of c_1 respectively c_2 are also constrained sets of $c_1 \wedge c_2$. Actually, if c_1 and c_2 do not share variables, these are the only ones. The hard case is when c_1 and c_2 do share variables. Consider a simple example in the numerical domain. Let c_1 be $x = y \wedge u = v$ and c_2 be $y + v = z$. Constrained sets of c_1 are $\{x, y\}$ and $\{u, v\}$; $\{y, v, z\}$ is the only constrained set of c_2. The conjunction $c_1 \wedge c_2$ entails constraints $x + v = z$, $y + u = z$ and $x + u = z$, giving rise to the constrained sets $\{x, v, z\}$, $\{y, u, z\}$, and $\{x, u, z\}$. At the concrete level, the key operation in obtaining entailed constraints is variable elimination. At the abstract level, the operation is mimicked by taking the union of an element of $close(AC_1)$ (which, to abstract c_1, must contain $\{x, y\}$, $\{u, v\}$, and $\{x, y, u, v\}$) and an element of $close(AC_2)$ (which, to abstract c_2, must contain $\{y, v, z\}$) and removing some elements from the intersection: removing y and v from $\{x, y, u, v\} \cup \{y, v, z\}$ yields $\{x, u, z\}$; removing y from $\{x, y\} \cup \{y, v, z\}$ yields $\{x, v, z\}$; and deleting v from $\{u, v\} \cup \{y, v, z\}$ yields $\{y, u, z\}$. Notice that one should not only remove the complete intersection, as shown by the following example. Consider $\{x, y, u, v\}$ as an element of AC_1 which abstracts, for example, $c_1 \equiv x + y = u + v$ and $\{x, y, t\}$ as an element of AC_2 which abstracts, for example, $c_2 \equiv x + y = t$ but also $c_2' \equiv x + 2y = t$. Now $c_1 \wedge c_2$ entails $t = u + v$ with constrained set $\{t, u, v\}$, while $c_1 \wedge c_2'$ entails $y = t - u - v$ and $x = 2u + 2v - t$ with constrained sets $\{t, u, v, y\}$ and $\{t, u, v, x\}$. This also illustrates that computing the abstraction of $c_1 \wedge c_2$ by abstract conjunction of the abstractions of c_1 and c_2 can be less precise than directly determining the constrained sets of the conjunction (which can be done by first transforming the conjunction to solved form).

In Dumortier [1994] it is shown how a similar reasoning applies for Herbrand constraints, PrologIII tuple constraints, and mixed constraints (over more than one constraint domain) and that abstract conjunction as defined below always yields a safe approximation (the proof is too long to be included here).

Definition 7.2.2 (*Abstract Conjunction*). Let $AC_1, AC_2 \in Cons^{\mathcal{F}^m}$. Then $AC_1 \wedge^{\mathcal{F}^m} AC_2 = min\mathcal{F}(AC_1 \cup AC_2 \cup (close(AC_1) \oplus close(AC_2)))$ where $SS_1 \oplus SS_2 = \{(S_1 \cup S_2) \setminus D \mid S_1 \in SS_1, S_2 \in SS_2, D \subseteq S_1 \cap S_2, D \neq \emptyset\} \setminus \{\emptyset\}$, and $close(AC)$ is the closure under union of AC.[17]

An equivalent but more efficient algorithm corresponding to Definition 7.2.2 is obtained by closing only the necessary parts of AC_1 and AC_2 (i.e., those parts containing common variables) and by taking care of not generating nonminimal sets when combining the two.

Let us now present how the abstract operations required by the framework are computed. One can take the same approach as in Section 6, defining AC_{entry}, AC^j_{in}, AC_{exit}, and AC' in terms of abstract projection $\exists^{\mathcal{F}^m}$, abstract conjunction $\wedge^{\mathcal{F}^m}$, and abstraction α^f. However, this results in a very poor precision. The reason is that it is disastrous to precision to add a numerical constraint to an abstract constraint store which already describes that constraint. For example, let c be the constraint $a_1x_1 + \ldots + a_nx_n = a_{n+1}$ and AC an abstract constraint store containing its abstraction, i.e., $\{x_1, \ldots, x_n\} \in AC$. Performing $\alpha^f(c) \wedge^{\mathcal{F}^m} AC$ creates an abstract constraint store AC' which includes the singleton $\{x_i\}$ for each of the variables x_i; hence, AC' indicates that each x_i is possibly nonfree. This computation reflects that AC abstracts an equation c', $b_1x_1 + \ldots + b_nx_n = b_{n+1}$. With an appropriate choice of values for $b_1, \ldots, b_n$, the constraint $c \wedge c'$ entails a constraint $dx_i = e$ which is abstracted as $\{\{x_i\}\}$, so it is required that $\{x_i\} \in AC'$. When *abstract_entry* passes an abstraction of a constraint to the entered procedure, then *abstract_exit* returns it, and the computation of AC' as suggested above destroys the freeness of all the involved variables.

To overcome this problem, we slightly revise the concrete semantics: a constraint c is represented as a pair (c_{old}, c_{new}) such that $c = c_{old} \wedge c_{new}$. The corresponding rewrite rules are:

—The c-transition (if consistent):
$S :: \langle c, G; (c_{old}, c_{new})\rangle \xrightarrow{c} S :: \langle c, G; (c_{old}, c_{new})\rangle :: \langle G; (c_{old}, c_{new} \wedge c)\rangle$.

—The r-transition:
$S :: \langle a, G; (c_{old}, c_{new})\rangle \xrightarrow{r} S :: \langle a, G; (c_{old}, c_{new})\rangle :: \langle b_1, \ldots, b_n; (\exists_{-vars(\rho)}(a = h \wedge c_{old} \wedge c_{new}), true)\rangle$
where $\rho : h\text{:-}b_1, \ldots, b_n$.

—The exit transition:
$S_1 :: \langle a, G; (c_o, c_n)\rangle \overset{\rho}{::} \langle b_1, \ldots, b_n; (c_{old}, true)\rangle :: S_2 :: \langle \Box_\rho; (c_{old}, c_{new})\rangle \xrightarrow{exit} S_1 :: \langle a, G; (c_o, c_n)\rangle \overset{\rho}{::} \langle b_1, \ldots, b_n; (c_{old}, true)\rangle :: S_2 :: \langle \Box_\rho; (c_{old}, c_{new})\rangle :: \langle G; (c_o, c_n \wedge \exists_{-vars(a)}(a = h \wedge c_{new}))\rangle$.

The modification of the exit transition is valid because $c_o \wedge c_n \wedge \exists_{-vars(a)}(a =$

[17] AC_1 and AC_2 are abstractions of sets of constraints, that are obtained by joining the abstractions of the individual constraints in the set (Definition 7.1.10). Thus, closing AC_1 and AC_2 at abstract conjunction implies that also constrained sets originating from different (independent) constraints are combined. This results in a possible loss of precision. The nonminimal freeness abstraction of Dumortier et al. [1993], however, exhaustively represents all combinations of constrained sets when abstracting each constraint (instead of computing these combinations at abstract conjunction) and hence prevents the loss of precision.

$h \wedge c_{old} \wedge c_{new}$) where $c_{old} = \exists_{-vars(\rho)}(a = h \wedge c_o \wedge c_n)$ is equivalent with $c_o \wedge c_n \wedge \exists_{-vars(a)}(a = h \wedge c_{new})$.

Now abstract constraint stores are also represented by a pair (AC_{old}, AC_{new}). The idea is that $(c_{old}, c_{new}) \in \gamma((AC_{old}, AC_{new}))$ iff $c_{new} \in \gamma(AC_{new})$ and $c_{old} \wedge c_{new} \in \gamma(AC_{old} \cup AC_{new})$. Reformulating the safety conditions of the framework for these modifications is a rather straightforward task and is omitted. The abstract operations can be defined as follows. With g a constraint and (AC_{old}, AC_{new}) its abstract call constraint, AC' is defined as $(AC_{old}, AC_{new} \wedge^{\mathcal{F}^m} \alpha^f(g))$. With g an atom, (AC_{old}, AC_{new}) its abstract call constraint, $\rho_1, \ldots, \rho_m$ the rules of the program P defining the predicate of g, and $\rho_j : h_j \text{:-} b_{j1}, \ldots, b_{jn_j}$, AC_{entry} is defined as $\exists^{\mathcal{F}^m}_{-vars(g)}(AC_{old} \cup AC_{new})$, and AC^j_{in} is defined as $(\exists^{\mathcal{F}^m}_{-vars(\rho_j)}(AC_{entry} \wedge^{\mathcal{F}^m} \alpha^f(g = h_j)), \emptyset)$. Finally, with $(AC^1_{old}, AC^1_{new}), \ldots, (AC^m_{old}, AC^m_{new})$ as the abstract out constraints of rules $\rho_1, \ldots, \rho_m$, AC_{exit} is defined as $upp(AC^1_{exit}, \ldots, AC^m_{exit})$ where $AC^j_{exit} = \exists^{\mathcal{F}^m}_{-vars(g)}(AC^j_{new} \wedge^{\mathcal{F}^m} \alpha^f(g = h_j))$, and AC' is defined as $(AC_{old}, AC_{new} \wedge^{\mathcal{F}^m} AC_{exit})$ and in *extension_from_table* as $(AC_{old}, AC_{new} \wedge^{\mathcal{F}^m} AC^{tab}\delta_{g=g^{tab}})$. Notice that AC_{entry}, AC_{exit}, and all entries in Table are not pairs but elements of $Cons^{\mathcal{F}^m}$ and that AC^j_{old} does not contribute to AC^j_{exit}. For further details, the reader is referred to Dumortier [1994].

Making the distinction between new and old information in the analysis of logic programs has been applied previously by Plaisted [1984] and by Mulkers [1993] and Mulkers et al. [1990; 1994].

Example 7.2.3 ($\mathcal{F}^m$ *Analysis for the sumlist Program*). The initial call pattern of $sumlist(A, B)$ is $\{\ \{A\}\ \}$, which is also the call pattern of the recursive call (the abstract information written out is the union of the old and new components of the compound abstract constraints).

sumlist(x, w) :-	% $\{\ \{x\}\ \}$
$\quad\{x = [\,]$,	% $\{\ \{x\}\ \}$
$\quad w = 0\}$.	% $\{\ \{x\}, \{w\}\ \}$
sumlist(x, w) :-	% $\{\ \{x\}\ \}$
$\quad\{x = [y \mid z]$,	% $\{\ \{x\}, \{y\}, \{z\}\ \}$
$\quad w = y + w'\}$,	% $\{\ \{x\}, \{y\}, \{z\}, \{w, w'\}\ \}$
$\quad$*sumlist(z,w')*.	% $\{\ \{x\}, \{y\}, \{z\}, \{w\}, \{w'\}\ \}$

The analysis indicates that, at the end of each rule, x and w are possibly nonfree. In the second rule, w and w' are free before the recursive call and depend on each other.

8. COMBINING THE TWO DOMAINS

The information inferred by the definiteness analysis and the freeness analysis of the previous two sections is enough to obtain a full mode system: the former provides modes **d** and **a** and the latter modes **f** and **a**. In a combination along the lines of Cousot and Cousot [1979] (applied in Codish et al. [1995]), the abstract domains and the original components of the basic operations remain the same, while during analysis interactions between the computed abstractions occur to refine them.

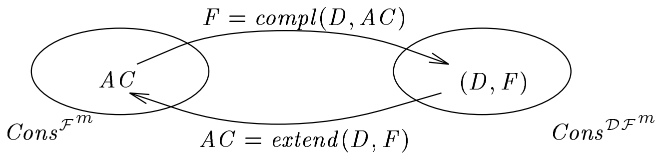

Fig. 2. Relation between $\mathcal{F}^m$ and $\mathcal{DF}^m$ abstraction (for a given D).

This results in a precise combined analysis, in particular when the analyses being composed contain a sufficient degree of "overlapping" information. As our domains are in a sense complementary, we present another kind of combination. Essentially, the D part of the definiteness analysis can be used as additional knowledge for the freeness abstraction. In this section we briefly present the improved freeness abstraction $Cons^{\mathcal{DF}^m}$ which is based on the minimal freeness abstraction and which uses additional knowledge about *definiteness* of program variables. In Section 9.1, we discuss how such an interaction between analyzers can be realized in a practical abstract interpretation system such as PLAI.

Definite variables occur in the minimal freeness abstraction as possibly nonfree variables. The presence of their corresponding singletons implies that the abstract operations have to take them into account — for example when computing the closure under union — although they play a very specific role in the propagation of possible nonfreeness. Efficiency of the analysis can be improved by separating out the definite variables. The assumption that the definite variables are known is reasonable, as the definiteness analysis computes a safe approximation (denoted by $defvars(\alpha^{\mathcal{D}}(C))$).

Given the set of definite variables D, $\alpha^{\mathcal{F}^m}(C)$ can be split into a set of singletons containing definite variables and a set of sets containing no definite variables, namely $compl(D, \alpha^{\mathcal{F}^m}(C)) = \{S \in \alpha^{\mathcal{F}^m}(C) \mid S \cap D = \emptyset\}$. The $\mathcal{DF}^m$ abstraction is based on the observation that the minimal freeness abstraction can be expressed in terms of $compl(D, \alpha^{\mathcal{F}^m}(C))$ and D (without loss of precision). The abstract domain $Cons^{\mathcal{DF}^m}$ is a set of pairs (D, F) where $D \subseteq Pvar$ and $F \in \wp(\wp_{\emptyset}(Pvar \setminus D))$ such that $min\mathcal{F}(F) = F$, to which $\perp$ is added as minimal element.

Definition 8.1 (*Abstraction of a Set of Constraints:* $\alpha^{\mathcal{DF}^m}$). Let $C \in Cons^{\mathcal{C}}$. Then $\alpha^{\mathcal{DF}^m}(C) = \perp$ if $C = \emptyset$; otherwise $\alpha^{\mathcal{DF}^m}(C) = (D, F)$ where D could be given by $defvars(\alpha^{\mathcal{D}}(C))$ and $F = compl(D, \alpha^{\mathcal{F}^m}(C))$.

There is a 1-to-1 correspondence between the abstractions in $Cons^{\mathcal{DF}^m}$ and $Cons^{\mathcal{F}^m}$ and vice-versa, for a given D (see Figure 2).

Definition 8.2 (*Extend*). Let $(D, F) \in Cons^{\mathcal{DF}^m}$. Then $extend(D, F) = F \cup \{\{x\} \mid x \in D\}$.

The operations on $Cons^{\mathcal{DF}^m}$ are based on the corresponding operations on $Cons^{\mathcal{D}}$ and $Cons^{\mathcal{F}^m}$. For their exact definitions we refer to Dumortier and Janssens [1994] and Dumortier [1994]. Concerning abstract conjunction of two abstract constraints (D_1, F_1) and (D_2, F_2), an efficient operation is obtained as follows: the D parts are joined first, and then the obtained definiteness information is propagated onto the freeness parts F_1 and F_2, thus reducing them considerably, before these are joined. Consequently, the $\mathcal{DF}^m$ abstract conjunction is much more efficient than if one would perform the $\mathcal{D}$ and $\mathcal{F}^m$ abstract conjunctions on the D and F parts separately, afterward deleting the definite variables from the resulting F part.

Again, as for the $\mathcal{F}^m$ analysis, an abstract constraint should be split into an old component, containing the information passed down from a calling environment, and a new component, containing the information that is gathered during local analysis of the rule body. Otherwise, too much precision would be lost at *abstract_exit*.

Example 8.3 ($\mathcal{DF}^m$ *Analysis for the sumlist Program*). The initial call pattern is *sumlist*$(\mathbf{d}, \mathbf{f})$, which is also the call pattern of the recursive call. The definiteness information is as in García de la Banda and Hermenegildo [1993]; we obtain the same freeness information as in Example 7.2.3, but in a more compact form (old and new components of the freeness part are put together).

```
sumlist(x, w) :-            % ({x}, ∅)
     { x = [ ],             % ({x}, ∅)
       w = 0 }.             % ({x, w}, ∅)
sumlist(x, w) :-            % ({x}, ∅)
     { x = [y | z],         % ({x, y, z}, ∅)
       w = y + w' },        % ({x, y, z}, {{w, w'}})
     sumlist(z, w').        % ({x, w, y, z, w'}, ∅)
```

9. EXPERIMENTAL RESULTS

In this section we present the results of the experiments that we have performed in order to evaluate the efficiency and accuracy of the analyses. We start by describing the implementation and the benchmarks used. Our attention then first focuses on the issue of efficiency and, thus, of the feasibility and scalability of the approach. This is an important issue, since it has been shown that even relatively simple analyses of LP programs have worst-case exponential behavior [Debray 1995]. On the other hand, it has also been shown experimentally that average-case behaviors have much better characteristics for typical analyses [Bueno et al. 1994; Debray 1992b; Le Charlier and Van Hentenryck 1994; Muthukumar and Hermenegildo 1992; Van Roy and Despain 1992; Warren et al. 1988]. It is obviously interesting to explore if this practical behavior carries over to our CLP analyses, both when analyzing CLP programs and when analyzing traditional LP programs (for comparison with LP analyzers). To study this point, we present a summary of the analysis times for a set of benchmarks which includes CLP programs (both relatively small and larger ones) and LP programs. The larger CLP programs are the largest programs available to us, and they should be instrumental in giving an idea of the scalability of the results in the new application area.

We then focus on the effect of an important technique related to the scalability issue: the application of widening operations in order to trade precision for efficiency. We investigate the effects on the efficiency and precision of our analyses of the introduction of widening in the freeness abstraction.

Finally, we perform a more detailed evaluation, focused on a representative set of CLP programs, in order to gain insight into the potential of the analyses, the main causes for loss of accuracy, and the advantages and disadvantages of the combined analyses. We do not address herein the obviously interesting issue of how the derived information can be used to optimize CLP programs, which we consider to be outside the scope of this article. However, and as mentioned previously, this subject has recently been addressed by several authors, and their results show that, if information from global analysis such as that obtained by our analyses is available, it can in fact be used to perform optimizations which result in significant speedups [Dumortier 1994; García de la Banda 1994; Jaffar and Maher 1994; Jaffar et al. 1992; Jørgensen et al. 1991; Marriott and Stuckey 1993; Marriott et al. 1994].

9.1 Implementation Issues

The abstract domains described in Sections 6, 7, and 8 have been implemented within the PLAI abstract interpretation system [Muthukumar and Hermenegildo 1990; 1992] which is an incarnation of the framework presented. The resulting analyses can deal with CLP(H,N) programs and with some of the PrologIII-specific features, namely tuples and size relations.

A few details of PLAI are worth mentioning, since they are instrumental in understanding the results obtained during our evaluation. PLAI in principle assumes finite abstract domains and analyzes each predicate for each distinct key (the pair $\langle a, AC_{entry} \rangle$). This implies that PLAI performs a quite detailed analysis and can obtain several annotations for the same predicate (versions). The current implementation allows the user to choose between obtaining a transformed program in which the different versions of the predicates appear explicitly and are each annotated with their corresponding inferred information or, alternatively, obtaining essentially the original program where predicates are annotated with the upper bound of the annotations of the different versions of that predicate. In our experiments the former approach was selected (exceptions are indicated).

It is important to note that the only modification that was needed for extending PLAI to CLP languages was the elimination of a "unifiability" test performed before executing the abstract entry function. This test is performed in the analysis of traditional LP languages in order to avoid analyzing rules whose head does not (syntactically) unify with the current subgoal. Naturally, the domain-dependent abstract functions had to be implemented and incorporated into the system, but almost all the existing implementation was reused. We argue that this strongly supports our claim regarding the practical usefulness of the approach that we propose, especially considering that, as we believe our measurements show, the resulting system can analyze reasonably sized programs in quite reasonable times.

Finally, the integration of the $\mathcal{DF}^m$ analyzer has been performed as follows. Since the $\mathcal{DF}^m$ analysis uses definiteness information provided by the $\mathcal{D}$ analysis, the $\mathcal{D}$ and $\mathcal{DF}^m$ analysis are executed in a coroutining fashion. At each point of the analysis (i.e., at the application of one of the higher-level abstract operations), the definiteness operation is called first. Afterward, the set of definite variables is extracted from the result of that operation and passed as an extra parameter to the freeness operation. If the definiteness operation results in the abstract constraint $\perp$, the freeness operation proceeds with $\perp$. Thus, information is always passed from the definiteness to the freeness analysis; information passing in the other direction is restricted to the passing of $\perp$ information: if a freeness operation yields $\perp$ where the preceding definiteness operation did not give $\perp$, the subsequent definiteness analysis continues with $\perp$ (thus computation of useless information is avoided).

The effect of such combination can be quite subtle. On the one hand, the efficiency (both in terms of memory and time) of the $\mathcal{DF}^m$ analyzer can be better than that of simply running both the $\mathcal{D}$ and $\mathcal{F}^m$ analyzers. This can be due to several factors. First, the potential reduction in the size of the $\mathcal{DF}^m$ abstractions can reduce the memory consumption, which in turn affects the analysis times. Second, reductions in the size of the abstract constraints can also reduce the cost of the abstract operations. Finally, the combination has a "loop-merging" effect — a single pass over the program is sufficient for $\mathcal{DF}^m$ instead of the two passes needed otherwise.

On the other hand, if one of the analyses requires more fixpoint iterations than the other, this may have a negative effect on the efficiency of the combined execution. If, for example, the definiteness analysis reaches the fixpoint first, the extra iterations will imply some unnecessary *table lookups*, projections, and extensions for this analyzer. If the freeness analyzer is the one who first reaches the fixpoint, the overhead may be more substantial. This is because part of the definiteness abstraction is included in the freeness abstraction, and therefore all abstract operations may be redone. Such extra iterations could be avoided by first performing the definiteness analysis by itself and then using the programs annotated by the definiteness analysis as input for the freeness analysis. The detailed evaluation for a subset of CLP programs discusses the interaction in depth.

9.2 Benchmarks

The global set of benchmarks used contains 29 CLP programs and 25 LP programs. The CLP programs solve typical CLP problems and include small to relatively large programs (i.e., programs with 1 to 50 predicates and with 2 to 110 rules). Part of them are taken from the CLP($\Re$) distribution, the PrologIII distribution, and from the CLP literature [Colmerauer 1990; Van Hentenryck 1989; Van Hentenryck and Ramachandran 1994]. Others have been obtained from the partners in the PRINCE ESPRIT project, from P. Van Hentenryck, and from the vendor of Prolog III and Prolog IV, PrologIA. We have also included a large collection of LP benchmarks, ranging from relatively simple to quite complex programs, which has been used previously in the literature to evaluate analyzers for LP programs [Codish et al. 1995; Mulkers et al. 1995]. The number of predicates in these benchmarks ranges from 1 to 79 and the number of rules from 2 to 187. Since all LP programs are also CLP programs, the latter set of benchmarks adds another dimension to the benchmark suite which allows us to expand our study of the scalability issue. A brief description of all the benchmarks is given in the Appendix. Here we include Table I and Table II which list properties of the benchmark programs to which the complexity of the analysis is related. The size of the programs is indicated by means of the number of user-defined predicates (Pr) and the number of rules (Rl). The recursiveness of the programs is indicated by means of the number of recursive predicates that are not tail-recursive (R), the number of tail-recursive predicates (TR), and the number of nonrecursive predicates (NR). Programs containing recursive predicates lead to a more complex analysis than nonrecursive programs, especially if they are not tail-recursive. The tables also list the maximum and average number of variables in the program rules (MaxV and AvgV). The number of variables in a

Table I. Properties of the CLP Benchmarks

Program	Pr	Rl	R	TR	NR	MaxV	AvgV
dnf	3	32	2	1	0	7	2.3
vecmat1	8	15	0	7	1	8	2.9
laplace1	2	4	0	2	0	12	6.0
fib	1	3	1	0	0	3	1.0
meal	6	11	0	0	6	6	0.9
listlength	1	2	0	1	0	4	2.0
sumlist	1	2	0	1	0	4	2.0
mining	25	50	4	10	11	18	2.5
power	18	42	0	9	9	19	3.3
rectangle	5	10	2	2	1	9	3.3
vecmat2	8	15	0	7	1	8	3.1
num	17	97	0	0	17	10	2.4
laplace2	2	4	0	2	0	16	10.8
sendmm	4	7	0	3	1	11	2.7
trap	4	5	0	1	3	9	6.4
runkut	4	5	0	1	3	9	6.2
mortgage1	1	2	0	1	0	5	4.0
mortgage3	1	2	0	1	0	5	4.0
mortgage2	1	2	0	1	0	5	4.0
bridge	29	90	0	14	15	13	1.6
color4	8	21	0	3	5	9	2.2
color4F	8	110	0	3	5	9	0.4
cutstock	50	77	3	19	28	21	3.9
magic	7	14	0	6	1	5	2.4
magicC	5	9	1	3	1	8	2.4
periodic	3	5	0	1	2	11	4.0
perm	11	20	0	7	4	6	2.6
triangle	34	47	0	5	29	24	6.3
warehouse	12	38	1	4	7	21	2.4

rule typically affects the size of the abstract constraints for the different program points in the rule, which in turn influences the cost of the abstract operations.

9.3 Efficiency Results

In order to get an idea of the feasibility of the analyses proposed in this article Table III and Table IV list the total analysis times for the CLP and the LP programs respectively. The figures include the time for garbage collection and stack shifts and are averaged over 10 runs. All measurements have been done on a SUN Sparc 2 using SICStus 2.1 with the "fastcode" option. "–" indicates that the analyzer did not produce a result (because it ran out of memory). The last column in Tables III and IV gives the ratio of time taken by the combined analysis $\mathcal{DF}^m$ to the sum of $\mathcal{D}$ and $\mathcal{F}^m$. "Inf" indicates that the combined analysis is definitely better, since in these cases $\mathcal{F}^m$ does not produce a result. The average of Table III does not take into account *laplace*1.

Table II. Properties of the LP Benchmarks

Program	Pr	Rl	R	TR	NR	MaxV	AvgV
akl	10	18	2	4	4	10	3.6
akl_old	7	12	0	4	3	10	3.6
ann	53	187	19	13	21	17	2.5
append	1	2	0	1	0	4	2.5
bid	22	53	0	7	15	7	2.2
boyer	28	138	3	1	24	8	2.3
browse	16	32	1	11	4	12	3.7
deriv	15	62	4	3	8	6	3.0
grammar	7	15	0	0	7	6	1.9
icomp	71	170	19	18	36	20	5.0
kalah	41	78	9	10	22	12	3.8
mapcolor	8	12	0	4	4	6	3.1
peephole	16	134	10	3	3	8	2.8
pg	10	18	0	6	4	10	3.6
plan	16	29	0	4	12	6	2.7
qplan	44	148	16	11	17	16	3.1
qsort	3	6	1	1	1	7	3.5
queens	5	9	0	4	1	5	2.4
rdtok	18	55	9	6	3	7	3.3
read	25	91	7	4	14	13	3.9
serialize	6	12	2	2	2	7	3.8
tarjan	37	90	12	14	11	20	4.9
vlok	46	137	0	17	29	12	2.6
vlokgr	46	137	0	17	29	12	2.6
witt	79	163	22	23	34	18	4.5

For most benchmarks the analysis times are acceptable. The average $\mathcal{D}$ and $\mathcal{DF}^m$ analysis times of the LP programs are better than for the CLP benchmarks. This is to be expected, since constraints in LP programs (unification constraints) are in general less complex than typical CLP constraints, leading to smaller constrained sets and smaller abstractions. Also, there is usually more definiteness information to be exploited: LP programs are frequently "generate-and-test," whereas CLP programs are often of the "constrain-and-generate" type, which implies that definiteness information is only derived toward the end of the program. For most programs (46 out of 54) the $\mathcal{F}^m$ analysis takes longer than the $\mathcal{D}$ analysis. This can be partly explained by the different natures of the abstractions. $\mathcal{D}$ propagates definiteness information and collects definite dependencies between nondefinite variables. $\mathcal{F}^m$ propagates possible nonfreeness and collects possible dependencies among all variables. The freeness analysis inherently has a larger time and space complexity than the definiteness analysis. But also the abstract query[18] which is analyzed for the benchmark programs plays an important role (e.g., laplace with $laplace1(\mathbf{d})$ and $laplace2(M)$ where M is a matrix of free variables, and mortgage with $mortgage1(\mathbf{a},\mathbf{a},\mathbf{a},\mathbf{a},\mathbf{f})$, $mortgage2(\mathbf{a},\mathbf{a},\mathbf{a},\mathbf{f},\mathbf{a})$, and $mortgage3(\mathbf{f},\mathbf{d},\mathbf{d},\mathbf{d},\mathbf{d})$).

[18] More details are in the Appendix.

Table III. Timings of the Analyses for the CLP Programs

Program	Analysis times (seconds)			Comparison
	$\mathcal{D}$	$\mathcal{F}^m$	$\mathcal{DF}^m$	$\mathcal{DF}^m/(\mathcal{D}+\mathcal{F}^m)$
dnf	0.960	5.158	3.492	0.57
vecmat1	0.116	0.438	0.298	0.54
laplace1	0.060	–	0.134	Inf
fib	0.044	0.082	0.092	0.73
meal	0.032	0.074	0.078	0.74
listlength	0.010	0.022	0.020	0.63
sumlist	0.014	0.028	0.020	0.48
mining	1.852	7.990	9.700	0.99
power	2.192	10.558	4.688	0.37
rectangle	6.960	3.172	11.184	1.10
vecmat2	0.324	0.544	0.928	1.07
num	0.976	1.680	2.000	0.75
laplace2	0.776	0.262	1.222	1.18
sendmm	1.378	3.198	4.154	0.91
trap	2.472	0.494	3.718	1.25
runkut	0.052	0.394	0.142	0.32
mortgage1	0.124	0.148	0.358	1.32
mortgage3	0.030	0.148	0.122	0.69
mortgage2	0.125	0.086	0.206	0.98
bridge	3.316	9.470	33.508	2.62
color4	0.226	1.346	0.922	0.59
color4F	0.416	1.798	1.684	0.76
cutstock	1.250	5.552	2.940	0.43
magic	0.172	0.406	0.346	0.60
magicC	0.522	0.290	0.906	1.12
periodic	0.748	0.320	1.430	1.34
perm	0.250	0.760	0.958	0.95
triangle	52.030	216.662	256.734	0.96
warehouse	1.044	0.742	1.834	1.03
Average	2.706	9.708	11.856	0.89

For some benchmarks the execution time of $\mathcal{F}^m$ becomes quite large (*triangle* and *tarjan*) or even infinite (*laplace*1). The combined $\mathcal{DF}^m$ analysis seems to offer a partial solution. The size of the $\mathcal{F}^m$ abstractions can be reduced (sometimes substantially) when definiteness information is available. For *laplace*1 and *tarjan*, $\mathcal{DF}^m$ has very good performance, but *triangle* does not really seem to benefit from the combination. The ratio $\mathcal{DF}^m/(\mathcal{D}+\mathcal{F}^m)$ shows that also for the other programs the $\mathcal{DF}^m$ analysis performs quite well. Due to the previously discussed interactions with the fixpoint computations $\mathcal{DF}^m$ sometimes introduces overhead, but this remains acceptable (see the programs with a ratio > 1). The average for the ratio $\mathcal{DF}^m/(\mathcal{D}+\mathcal{F}^m)$ is 0.89 for the CLP programs (not taking into account *laplace*1) and 0.37 for the LP programs. The execution times for $\mathcal{DF}^m$ vary between 0.020 and 256.7 seconds.

Table IV. Timings of the Analyses for the LP Programs

Program	Analysis times (seconds)			Comparison
	$\mathcal{D}$	$\mathcal{F}^m$	$\mathcal{DF}^m$	$\mathcal{DF}^m/(\mathcal{D}+\mathcal{F}^m)$
akl	0.264	2.850	0.654	0.21
akl_old	0.182	2.626	0.454	0.16
ann	6.944	6.936	14.754	1.06
append	0.018	0.108	0.030	0.24
bid	0.348	1.304	0.708	0.43
boyer	3.850	9.822	14.756	1.08
browse	1.004	1.508	1.912	0.76
deriv	1.174	2.938	1.774	0.43
grammar	0.044	0.172	0.122	0.56
icomp	6.914	30.480	33.512	0.90
kalah	1.002	4.374	1.700	0.32
mapcolor	0.554	33.490	31.444	0.92
peephole	2.658	14.382	10.510	0.62
pg	0.192	0.868	0.386	0.36
plan	0.222	1.544	0.560	0.32
qplan	1.262	17.060	3.376	0.18
qsort	0.096	0.312	0.274	0.67
queens	0.066	0.284	0.140	0.40
rdtok	1.740	4.434	4.616	0.75
read	2.162	9.818	4.222	0.35
serialize	0.784	0.966	1.742	1.00
tarjan	1.900	124.340	6.126	0.05
vlok	1.236	34.452	3.886	0.11
vlokgr	0.964	33.826	2.688	0.08
witt	2.354	17.376	4.552	0.23
Average	1.517	14.251	5.796	0.37

9.4 Effects of Widening

As mentioned before, we have also studied the effect of the application of widening operations in order to trade precision for efficiency, which is an important technique related to the scalability issue. For the current set of benchmarks, scalability seems to be a problem of the freeness abstraction but not of the definiteness abstraction. Therefore, we decided to apply widening in the freeness analysis and in the freeness part of the $\mathcal{DF}^m$ analysis. The idea is to avoid the *close* operation (used for example during abstract conjunction) for large freeness abstractions, as in those cases this operation is too expensive. An abstraction is considered to be too large if it contains a number of nonsingleton sets above some bound. We experimented with two different bounds B: 10 (referred to as wid10 in Table V) and 8 (wid8 in Table V). If an abstraction contains B or more nonsingletons, widening is applied. A strong form of widening was used: that all variables involved in the abstraction are given mode "any."

Table V indicates the influence of widening on the timings and precision of the $\mathcal{F}^m$ analysis and freeness part of the $\mathcal{DF}^m$ analysis. As mentioned before, the column "Wid" indicates whether the analyzer includes widening or not and, if so, what the bound is on the number of nonsingleton sets in the abstraction ("wid10"

Table V. Widening Information

Program	Wid	Free	Time $\mathcal{F}^m$	Time $\mathcal{DF}^m$	Mem $\mathcal{F}^m$	Mem $\mathcal{DF}^m$
laplace1	nowid	0	–	0.134	–	4.298
	wid10	0	0.890	$\equiv$	4.298	$\equiv$
	wid8	0	0.610	$\equiv$	4.298	$\equiv$
mining	nowid	143	7.990	9.700	4.888	4.950
	wid8	143	2.500	4.660	4.829	4.892
power	nowid	191	10.558	4.688	4.888	4.856
	wid8	191	3.610	$\equiv$	4.829	$\equiv$
rectangle	nowid	18	3.172	11.184	4.794	4.825
	wid8	18	2.360	10.250	4.798	4.798
sendmm	nowid	66	3.198	4.154	4.888	4.888
	wid10	57	0.310	1.670	4.298	4.798
	wid8	57	0.290	1.710	4.298	4.798
bridge	nowid	204	9.470	33.508	4.888	6.075
	wid8	204	5.490	9.380	4.829	4.892
triangle	nowid	1664	216.662	256.734	13.263	13.950
	wid10	1508	44.250	106.370	6.392	6.329
	wid8	1508	6.450	98.710	5.204	6.329
tarjan	nowid	439	124.340	6.126	6.138	5.075
	wid10	439	89.280	$\equiv$	6.142	$\equiv$
	wid8	439	32.020	$\equiv$	6.142	$\equiv$
vlok	nowid	216	34.452	3.886	8.513	5.138
	wid10	216	13.370	$\equiv$	5.142	$\equiv$
	wid8	216	8.320	$\equiv$	5.142	$\equiv$
vlokgr	nowid	216	33.826	2.688	8.513	5.138
	wid10	216	13.400	$\equiv$	5.142	$\equiv$
	wid8	216	8.440	$\equiv$	5.142	$\equiv$

or "wid8"). The column "Free" indicates the number of free-variable annotations derived by the analysis.[19] It shows when precision is lost due to widening. The following two columns contain the timings (in seconds) for the $\mathcal{F}^m$ and $\mathcal{DF}^m$ analyses respectively. The last two columns indicate the memory consumption (in megabytes, giving maximum amount of memory allocated by the UNIX system to the PLAI process). The table contains only those benchmarks where widening is actually applied: for wid8, 10 out of the 54 benchmarks effectively apply widening in the case of $\mathcal{F}^m$, and 5 in the case of $\mathcal{DF}^m$. For wid10, widening happens only in, respectively, 6 and 2 benchmarks. "$\equiv$" indicates that widening is not triggered for the particular program and analysis (time and memory figures then correspond to the nowid figures). Notice that widening is not triggered in the case of the $\mathcal{DF}^m$ analysis of the LP benchmarks. Applying the widening operation (wid8) allows analyzing the *laplace*1 benchmark in 0.610 seconds using 4.298MB, where the original $\mathcal{F}^m$ analysis (without widening) did not produce a result within reasonable time and memory bounds (indicated by "–" in the table).

[19] These figures are the same for the $\mathcal{F}^m$ and $\mathcal{DF}^m$ analysis, and they are computed selecting the analysis output option that returns an annotated version of the original program with each predicate annotated with the upper bound of the analysis results for all the different entry-exit patterns (the different version) inferred during the analysis.

In general, if widening is applied it improves considerably the execution times and memory consumption. For some programs (*sendmm* and *triangle*) the difference is an order of magnitude. The maximum analysis time for $\mathcal{F}^m$ (nowid) is 216.7 sec. for *triangle*, and with wid8 it goes down to 6.5 sec., while for $\mathcal{DF}^m$ it goes down from 256.7 to 98.7 sec. The impact of widening is not the same for $\mathcal{F}^m$ and $\mathcal{DF}^m$. In the case of *triangle* this is due to the $\mathcal{D}$ part of the analysis. For the $\mathcal{D}$ analysis, the execution time of *triangle* (52 sec.) differs one order of magnitude with respect to the other execution times. The $\mathcal{D}$ analysis infers large definite dependency sets, since *triangle* uses the CLP "constrain-and-generate" programming technique. A widening for the $\mathcal{D}$ analysis could lessen this kind of inefficiencies.

The effect on the precision is acceptable, as only in two cases (*sendmm* and *triangle*) precision is lost when considering for each predicate the single upper bound which is computed from the analysis results for the different entry-exit patterns. These experiments suggest that widening allows to avoid exponentional time and memory consumption and to keep the loss of precision very small.

9.5 A More Detailed Evaluation

In addition to the more general study reported above, in order to gain more insight into the behavior of the analyses in typical CLP programs we performed a more detailed evaluation on a subset of the benchmarks (namely, the first 19 CLP programs of Table I), which we consider a representative selection of typical CLP programs.

9.5.1 *A Closer Look at the Efficiency Results: Measurements.* Our aim is (1) to study the time and memory consumption of each of the analyzers (only taking into account the kernel of the analysis, i.e., the execution of the fixpoint algorithm, and not the pre- and postprocessing phase) and (2) to evaluate the effectiveness of the combined analysis with respect to the individual $\mathcal{D}$ and $\mathcal{F}^m$ analyses. For this purpose the following figures have been collected:

—Regarding the fixpoint computation: the number of entry-exit patterns for all predicates (EE) and for the recursive predicates only (EEr), and the number of fixpoint iterations (FIX). They are presented in Table VI. These numbers largely determine the complexity of the analyses and will be used to explain the time and memory figures of the combined analysis, when compared to those of the individual $\mathcal{D}$ and $\mathcal{F}^m$ analyses. Table VI also lists the number of syntactically different calls modulo renaming (DCls), which is a lower bound on the number of entry-exit patterns that will be computed by the analyzer under the assumption that the program does not have dead code.

—Regarding time consumption: the total analysis times (including the time for garbage collection and stack shifts) averaged over 10 runs. They are given in Table III, the last column provides the execution time comparison.

—Regarding memory consumption:

(1) the maximal amount of memory allocated by the UNIX system to the PLAI process. It falls between 4.293808MB and 4.918808MB (except for the $\mathcal{F}^m$ analysis of *laplace*1 which runs out of memory). Note that the sum of the memory allocated by the $\mathcal{D}$ and the $\mathcal{F}^m$ analyzers is always larger than the memory allocated by $\mathcal{DF}^m$;

Table VI. Number of Fixpoint Iterations and Entry-Exit Patterns

Program	DCls	$\mathcal{D}$ EE	$\mathcal{D}$ EEr	$\mathcal{D}$ FIX	$\mathcal{F}^m$ EE	$\mathcal{F}^m$ EEr	$\mathcal{F}^m$ FIX	$\mathcal{DF}^m$ EE	$\mathcal{DF}^m$ EEr	$\mathcal{DF}^m$ FIX
dnf	14	14	14	14	26	26	26	26	26	26
vecmat1	9	9	8	8	11	10	13	11	10	10
laplace1	4	4	4	4	-	-	-	4	4	4
fib	3	3	3	3	3	3	3	3	3	3
meal	6	6	0	0	6	0	0	6	0	0
listlength	1	1	1	1	1	1	1	1	1	1
sumlist	1	1	1	1	1	1	1	1	1	1
mining	32	36	25	39	32	20	28	36	25	39
power	24	28	19	27	24	15	18	28	19	27
rectangle	8	11	10	21	9	8	14	11	10	22
vecmat2	10	12	11	16	13	12	15	16	15	22
num	17	20	0	0	17	0	0	20	0	0
laplace2	3	3	3	4	3	3	4	3	3	4
sendmm	6	6	5	7	6	5	6	6	5	7
trap	5	11	4	5	5	2	3	11	4	5
runkut	6	6	1	1	6	1	2	6	1	2
mortgage1	2	2	2	2	3	3	3	3	3	3
mortgage3	2	2	2	2	3	3	3	3	3	3
mortgage2	2	2	2	2	2	2	2	2	2	2

(2) the total amount of global stack space (Glob) and program space (Prog)[20] used during the actual analysis. This is given in Table VII. The last column in Table VII compares the sum of the global stack and program space used by the $\mathcal{DF}^m$ analysis with the maximum of the sum of global stack and program space used in the $\mathcal{D}$ and the $\mathcal{F}^m$ analyses, i.e., the memory consumption[21] comparison figure.

In order to aid in the interpretation of the results we divide the programs into two classes:

(1) Programs that, for the given entry patterns, constrain many variables from the start to definite values and related dependencies (dnf, vecmat1, laplace1, fib, meal, listlength, and sumlist).
(2) Programs that do not allow inferring much definiteness information or where it is inferred only toward the end of the program (mining, power, rectangle, vecmat2, num, laplace2, sendmm, trap, runkut, mortgage1, mortgage3, and mortgage2). They create and handle large sets of possible dependencies.

In each class the benchmarks are ordered starting with the highest estimate for the size of the AND-OR graph.[22]

[20] The global stack stores the compound terms. Program space refers to the amount of memory allocated for compiled and interpreted rules, symbol tables, the record database, and the like.
[21] In what follows, we refer to the global stack and program space consumption simply as memory consumption.
[22] We use the formula $(Rl/Pr) * AvgV * (NR + FIX * (TR + 3 * R))$ with FIX of $\mathcal{DF}^m$ given in Table VI and the rest in Tables I and II.

Table VII. Global Stack Space and Program Space Used During Analysis (in megabytes)

Program	$\mathcal{D}$ Glob	$\mathcal{D}$ Prog	$\mathcal{F}^m$ Glob	$\mathcal{F}^m$ Prog	$\mathcal{DF}^m$ Glob	$\mathcal{DF}^m$ Prog	Comparison $\mathcal{DF}^m/max(\mathcal{D},\mathcal{F}^m)$
dnf	0.303	0.065	1.737	0.133	1.081	0.149	0.66
vecmat1	0.049	0.008	0.194	0.011	0.110	0.013	0.60
laplace1	0.033	0.004	–	–	0.060	0.006	Inf
fib	0.022	0.003	0.045	0.003	0.040	0.004	0.92
meal	0.018	0.004	0.041	0.004	0.038	0.005	0.96
listlength	0.014	0.001	0.019	0.001	0.017	0.001	0.90
sumlist	0.014	0.001	0.021	0.001	0.017	0.001	0.82
mining	0.805	0.054	1.312	0.055	2.121	0.083	1.61
power	1.062	0.063	2.151	0.052	1.825	0.090	0.87
rectangle	3.054	0.019	0.856	0.016	4.127	0.028	1.35
vecmat2	0.147	0.012	0.216	0.014	0.401	0.022	1.84
num	0.216	0.064	0.532	0.061	0.541	0.084	1.05
laplace2	0.444	0.008	0.105	0.006	0.535	0.011	1.21
sendmm	0.484	0.012	0.792	0.010	1.250	0.017	1.58
trap	1.280	0.017	0.198	0.007	1.629	0.024	1.27
runkut	0.026	0.005	0.145	0.007	0.061	0.008	0.45
mortgage1	0.076	0.002	0.092	0.004	0.217	0.004	2.30
mortgage3	0.020	0.002	0.092	0.004	0.064	0.004	0.71
mortgage2	0.076	0.002	0.063	0.003	0.135	0.003	1.77

Evaluation. For the first class of programs, the time and memory figures correspond quite well with the complexity estimate used for ordering them. The number and the size of the dependencies is small and hence has not much influence on the figures. For the second class of programs, however, the complexity estimate is no longer adequate to predict the time and memory consumption. In this case, the number and the size of the dependencies can have a more important impact. Note that for the $\mathcal{F}^m$ analysis, the programs with large time and memory figures (*power*, *mining*, *sendmm*, and *rectangle*) have relatively many variables in their rules (i.e., large MaxV and AvgV numbers in Tables I and II). This trend can also be observed for the $\mathcal{D}$ analysis (*rectangle*, *trap*, *power*, *mining*, and *sendmm*). This trend is also confirmed by the actual output of the analysis and by the correlation between analysis time and global stack consumption, since the latter is dominated by the size of the abstract constraints built during the analysis.

The $\mathcal{DF}^m$ analysis yields quite satisfactory results for the considered benchmarks, both concerning time and memory consumption. The execution times vary between 0.020 and 11.184 seconds. For most benchmarks (14 out of 19), the execution time comparison figure of Table III is smaller than 1. Also, for 10 of the 19 benchmarks the memory consumption comparison figure of Table VII is smaller than 1, and only for one benchmark it is larger than 2. This provides evidence that combining the $\mathcal{D}$ and $\mathcal{F}^m$ analyzers indeed results in a practical full mode analysis system. We now perform a more detailed evaluation of these figures, based upon the classes of programs and their complexity in terms of entry-exit patterns and fixpoint iterations (Table VI).

As mentioned before, the first class of programs yields many definite variables right at the beginning of the execution. For these programs, the definiteness information is effectively used in the freeness part. This is reflected both in the timings (upper part of Table III) and the memory consumption (upper part of Table VII). In some cases (*dnf*, *vecmat*1), the $\mathcal{D}$ analyzer has to iterate along with the $\mathcal{F}^m$ analyzer when combining the two, i.e., the FIX and EE numbers of the $\mathcal{DF}^m$ analyzer correspond to the ones of the $\mathcal{F}^m$ analyzer and are larger than those of the $\mathcal{D}$ analyzer (Table VI). But even then, this overhead is more than compensated by the benefit of exploiting definiteness information, so there is still a considerable improvement of $\mathcal{DF}^m$ with respect to $\mathcal{D} + \mathcal{F}^m$.

For the second class of programs, the combination does not always pay off. Clearly, it depends on whether or not the gain obtained by exploiting definiteness information in the freeness part outweighs the overhead caused by extra fixpoint iterations (FIX) and entry-exit patterns (EE) in $\mathcal{DF}^m$ compared to $\mathcal{D}$ and/or $\mathcal{F}^m$. Four situations can be distinguished concerning the FIX and EE numbers of the $\mathcal{DF}^m$ analysis with respect to those of $\mathcal{D}$ and $\mathcal{F}^m$. First of all, for the *mining*, *power*, *num*, *rectangle*, *trap*, and *sendmm* programs, the EE and FIX figures for the $\mathcal{F}^m$ analyzer are smaller than the ones for the $\mathcal{D}$ analyzer. Thus, when combining the two analyses, the $\mathcal{DF}^m$ analysis has to perform at least as many iterations as the $\mathcal{D}$ analysis. However, it now not only computes the definite part, but it also takes into account the (reduced) freeness part. In the case of *mining*, *power*, *sendmm*, and *num*, this overhead is outweighed by the gain obtained from exploiting definiteness information (the execution time comparison figures are smaller than 1, and the memory consumption comparison figures fall between 0.87 and 1.61), whereas for *rectangle* and *trap*, the freeness part cannot benefit much from the definiteness information (note that in those cases the time and memory consumption for the $\mathcal{D}$ analysis is large — both by itself and compared with the $\mathcal{F}^m$ analysis — which indicates that mostly definite dependencies are derived rather than definite variables). Second, for the *runkut*, *mortgage*1, and *mortgage*3 benchmarks, the EE and FIX numbers of the $\mathcal{DF}^m$ analysis correspond to those of the $\mathcal{F}^m$ analysis and are larger than the $\mathcal{D}$ ones. In the case of *runkut* and *mortgage*3, the time and memory figures show that the freeness part can benefit quite well from the definite information. Also, the $\mathcal{D}$ time and memory consumption is small compared to that of $\mathcal{F}^m$, so the extra iterations of $\mathcal{D}$ (when forced to execute along with $\mathcal{F}^m$) are not outweighing the gain. For *mortgage*1 however, the situation is just the opposite: the execution time comparison figure is larger than 1, and the memory consumption comparison figure is larger than 2. Third, the *laplace*2 and *mortgage*2 benchmarks have the same EE and FIX numbers for all analyses. Although there are no extra iterations, there is almost no definiteness information to be exploited. The combination may cause a slight overhead due to the extra operations dealing with the (in this case useless) communication between the two analyses (cf. time for *laplace*2). Finally, for the *vecmat*2 benchmark, the $\mathcal{DF}^m$ analysis performs more iterations and has more entry-exit patterns than either one of the $\mathcal{D}$ and $\mathcal{F}^m$ analyses. This results in a slight overhead in memory consumption and analysis time.

9.5.2 *Accuracy Results: Measurements.* The accuracy of the analyzers is determined by comparing the outcome of concrete executions of the benchmarks with the results obtained by the analyses. More precisely, the correct (concrete) modes of the variables at each program point are compared with the modes derived by

Table VIII. Accuracy of the Analyzers (only w.r.t. variable modes)

Program	Annot	ImpD	ImpF	PrecD	PrecF	PrecD+F	Imp. cause
dnf	3772	0	33	100.0	99.1	99.1	(1)
vecmat1	125	0	6	100.0	95.2	95.2	(2)
laplace1	112	0	0	100.0	100.0	100.0	
fib	12	0	0	100.0	100.0	100.0	
meal	56	0	0	100.0	100.0	100.0	
listlength	12	0	0	100.0	100.0	100.0	
sumlist	12	0	0	100.0	100.0	100.0	
mining	1064	105	80	90.1	92.5	82.6	(1)
power	1256	126	73	89.9	94.2	84.1	(1,2)
rectangle	343	0	4	100.0	98.8	98.8	(1)
vecmat2	208	0	13	100.0	93.7	93.7	(1,2)
num	1402	13	0	99.1	100.0	99.1	
laplace2	124	0	60	100.0	51.6	51.6	(1)
sendmm	141	0	0	100.0	100.0	100.0	
trap	135	73	8	46.0	94.0	40.0	(2)
runkut	119	0	14	100.0	88.2	88.2	(2)
mortgage1	54	0	8	100.0	85.0	85.0	(2)
mortgage3	36	3	2	91.6	94.4	86.0	(2)
mortgage2	36	0	0	100.0	100.0	100.0	
Average				95.6	94.0	89.6	

the analyzers. If specialized versions of a predicate arise during concrete execution, these are considered separately. The predicate versions produced by the analyzers are mapped onto the concrete versions (usually, there is a one-to-one correspondence between the concrete and abstract predicate versions; in some cases however, several abstract versions map onto one concrete version or vice-versa). The figures for the $\mathcal{DF}^m$ analysis are presented in Table VIII (a similar study could be made for the individual $\mathcal{D}$ and $\mathcal{F}^m$ analyses which may infer a different number of predicate versions; herein we approximate these figures by considering the $\mathcal{D}$ part and the $\mathcal{F}^m$ part of the combined $\mathcal{DF}^m$ analysis separately). Column "Annot" gives the total number of variable annotations (summed up over the predicate versions and the program points). "ImpD" and "ImpF" give the number of imprecise variable annotations (derivation of mode **a** instead of **d**, respectively mode **a** instead of **f**). The columns "PrecD" and "PrecF" give the percentages of variable modes that are correctly inferred by the $\mathcal{D}$ part of the analysis and the $\mathcal{F}^m$ part respectively. "PrecD+F" is the percentage of correct variable modes derived by the combined $\mathcal{DF}^m$ analysis.[23] The average precision is shown at the bottom of the table. The last column indicates the cause of imprecision.

[23]This number is lower than or equal to the corresponding PrecD and PrecF number, as it takes into account *both* imprecision due to deriving mode **a** instead of **d** and that due to deriving mode **a** instead of **f**, whereas in the $\mathcal{D}$ part only imprecision of the type "mode **a** instead of **d**" is taken into account (as mode **a** is the most precise abstraction for free variables in the $\mathcal{D}$ analysis), and since in the $\mathcal{F}^m$ part only imprecision of the type "mode **a** instead of **f**" is taken into account (as mode **a** is the most precise abstraction for definite variables in the $\mathcal{F}^m$ analysis).

Besides the accuracy of mode annotations, one can additionally consider the accuracy of the dependency information.[24] In the case of possible dependencies, the same precision is obtained with the $\mathcal{F}^m$ and $\mathcal{DF}^m$ analyzers. Even if correct modes are inferred at a particular program point, the inferred possible dependencies may not occur in the concrete case or may be too strong compared to the concrete dependencies, thus possibly leading to imprecise mode annotations at subsequent program points. Imprecise dependency information not affecting the precision of the mode information (not visible in Table VIII) is derived when analyzing the *sendmm* program (about 40% of the dependencies are too strong) and, to a lesser extent, also in the *power* and *runkut* benchmarks.

Evaluation. The average precision for the $\mathcal{D}$ part is 95.6%, 94% for the $\mathcal{F}^m$ part, and 89.6% for the combined $\mathcal{DF}^m$ analyzer. For the $\mathcal{D}$ part and the combined $\mathcal{DF}^m$ analysis, the worst case occurs for the *trap* benchmark (respectively 46% and 40%). For the $\mathcal{F}^m$ part, the worst results are for the *laplace*2 benchmark (51.6%).

There are three sources of inaccuracy: (1) the lack of information about term structures, (2) the treatment of nonlinear constraints, and (3) the abstraction of primitive constraints instead of the abstraction of conjunctions of primitive constraints. The first is mainly related to inaccuracy of modes. The third mainly affects the accuracy of the dependency information. The second influences both.

Regarding the lack of information about term structure, when selecting a component of a partially instantiated term having mode **a**, the definiteness analysis cannot discover the definiteness of a definite subterm. Similarly, the freeness analysis cannot recognize free variables within the term. Consider a program scheme of the form

$$build\_structure(Data), constrain(Data), instantiate(Data)$$

where one first builds a data structure, then imposes constraints on that structure, and finally instantiates it. Such a scheme is used quite frequently within CLP (e.g., *mining*, *power*, *rectangle*, *sendmm*,...). It gives rise to a loss of precision when selecting components of the structure within *constrain*/1 and *instantiate*/1. Imprecision due to the absence of structure information occurs in case of the *dnf*, *rectangle*, *laplace*2, and *mining* benchmarks and is causing part of the imprecision in *power* and *vecmat*2. Although adding structure information [Janssens and Bruynooghe 1992; Le Charlier and Van Hentenryck 1994; Mulkers et al. 1995] could clearly improve precision, it also complicates the analysis in the sense that, when changing the abstract representation for the unification part, the interaction between the unification and numerical part has to be revised. The second source of imprecision is the abstraction of nonlinear constraints. This is the cause of inaccuracy in *runkut*, *trap*, *mortgage*1, *mortgage*3, *vecmat*1 and partly in *power* and *vecmat*2. Finally, in the *sendmm* benchmark, imprecise possible dependency information is derived due to abstracting primitive constraints and joining their abstraction via abstract conjunction, instead of abstracting a conjunction of primitive constraints at once. In theory, loss of precision (at least for the freeness part) is also possible due to the imprecise abstraction of disequations and inequalities. However, it did not occur

[24] We only consider the *possible* dependency information. A similar study could be made for the definite dependencies.

in the benchmarks considered. Also, minimization could lead to loss of precision, by combining via union dependencies that are unrelated (i.e., which result from different OR branches in the computation). Note that this is not as bad as applying transitivity on dependency relations, as is done for some LP mode analyses, but it may nevertheless lead to imprecise results. Again, no such imprecision was found for the benchmarks. It might be argued that, in practical programs, different predicate rules usually establish the same or comparable dependencies among the variables of a call to the predicate.

9.6 Conclusion

The detailed experimental evaluation provides good insight regarding the potential efficiency and accuracy of the analyzers, the main causes for loss of accuracy, and the advantages and disadvantages of the combined analyzer. It shows that the combination of the $\mathcal{D}$ and $\mathcal{F}^m$ analyzers results in a practical full mode analysis system. Moreover, our experiments indicate that the analyses scale up quite well for larger programs. Problems — if any — have not so much to do with the size of the program but with the number of variables in a clause and can be overcome with the use of a widening operator. Our results provide evidence of the feasibility of abstract interpretation as a powerful tool for the analysis of CLP programs.

10. CONCLUSIONS AND DISCUSSION

The generalization of analysis frameworks for logic programs (based on abstract interpretation) has been presented as a practical approach to the dataflow analysis of constraint logic programming languages. In particular, we have proposed an extension of Bruynooghe's traditional framework which allows it to analyze constraint logic programs. Using this generalized framework, two analyses have been proposed for approximating definiteness and freeness information respectively, as well as a combined analysis inferring both properties. We have also reported on the implementations of the framework and the domains and on the study of these implementations. Finally, we have shown that simple widening operators are adequate for controlling the analysis time of large or complex programs. The experimental results support our claim that with the approach proposed it is possible to obtain practical, accurate, and efficient analyses, while reusing much of the framework technology developed for traditional logic programming.

We believe that, given the adaptability of traditional frameworks to CLP analysis, future work might concentrate on accurately approximating the new properties needed for effectively applying the different optimizations relevant to the CLP paradigm. Encouraging examples in this direction are García de la Banda et al. [1993], Marriott and Stuckey [1994], and Macdonald et al. [1993]. The difficulties in this task come from many sources. First, it requires a good abstraction of (possibly many) constraint solver algorithms which are typically more complex than the well-known unification. This in turn implies abstracting enough information for simulating the way in which the solver propagates the property of interest. This information seems to be closely related to the abstraction of the entailment relation. The problem is then to determine which constraints from all those entailed are relevant to the property being abstracted. It is interesting to note how correctness problems encountered by early analyzers for LP in the context of variable "aliasing" can be reinterpreted in this context. After analyzing the goals $X = Y$, $Y = Z$, and $Z = a$, X can be inferred (incorrectly) to be a free variable or (inaccurately) to be $\top$. This problem can now be seen as related to not taking into account the entailed relation $X = Z$ which is relevant to the propagation of nonfreeness and groundness information.

Second, most 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. Therefore, one must take into account the effects that the conjunction of a particular constraint can produce with respect to any of the other constraint systems in the language. The abstract domains proposed in this article handle this directly. However, it may be preferable to be able to specify the abstraction for each constraint domain separately and then deal with the interactions. This suggests organizing the domains and analyses as a hierarchy 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 would be used for performing the transfer of information among the lower-level domains that is necessary in order to preserve correctness and achieve reasonable efficiency. Alternatively, rather than having a top-level domain, transfer functions among all domains can be specified. A negative aspect of the separation of domains and of the explicit interaction among them is that the same information could be represented several times. Also, for some abstractions, it could be difficult to define interaction rules such that there is no loss of precision.

Finally, and from a practical point of view, one must consider the vehicle to be used for implementing the abstract operations. As mentioned before, Codognet and Filé [1992] 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, one favorable aspect of formulating analyses so that they can be executed using only equalities over the Herbrand domain is generality: 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.

APPENDIX

BENCHMARK PROGRAMS

The CLP benchmark programs solve typical CLP programs. A representative subset of the CLP programs are used in our detailed experiments. The programs solve typical CLP problems. Most of them are taken from the CLP($\Re$) distribution, the PrologIII distribution, or the PRINCE project benchmarks. For this subset, we specify the abstract query. This query is given in the simplified "mode" format available to the user. Modes **d**, **f**, and **a** mean that the argument is definite, free, or any term respectively. This specification is translated into the appropriate representation for each domain.

—**dnf:** converts a propositional formula into disjunctive normal form; entry pattern $dnf(\mathbf{d}, \mathbf{f})$.
—**fib:** $fib(N, F)$ expresses that F is the Nth Fibonacci number; entry pattern $fib(\mathbf{d}, \mathbf{f})$.
—**laplace:** solves the Dirichlet problem for Laplace's equation in two dimensions using Leibman's five-point finite-difference approximation; entry patterns $laplace1(\mathbf{d})$ and $laplace2(M)$ where M is a matrix of free variables.
—**listlength:** specifies the relation between a list and its length; entry pattern $listlength(\mathbf{d}, \mathbf{f})$.
—**meal:** computes a balanced meal; entry pattern $lightMeal(\mathbf{f}, \mathbf{f}, \mathbf{f})$.
—**mining:** optimizes the revenue of an open mine; entry pattern $mining(\mathbf{f}, \mathbf{f})$.
—**mortgage:** well-known mortgage program; entry patterns $mortgage1(\mathbf{a}, \mathbf{a}, \mathbf{a}, \mathbf{a}, \mathbf{f})$, $mortgage2(\mathbf{a}, \mathbf{a}, \mathbf{a}, \mathbf{f}, \mathbf{a})$, and $mortgage3(\mathbf{f}, \mathbf{d}, \mathbf{d}, \mathbf{d}, \mathbf{d})$.
—**num:** transforms numbers into a sequence of letters and phonemes; entry pattern $nombre(\mathbf{d}, \mathbf{f}, \mathbf{f})$.
—**power:** minimizes the production cost of power stations; entry pattern $pow(\mathbf{f})$.
—**rectangle:** fills a rectangle with squares; entry pattern $fillRectangle(\mathbf{f}, \mathbf{a})$.
—**runkut:** first-order ordinary differential equation solving, using the Runge-Kutta method; entry pattern $solve(\mathbf{d}, \mathbf{d}, \mathbf{f})$.
—**sendmm:** $send + more = money$ puzzle; entry pattern $solution(\mathbf{f}, \mathbf{f}, \mathbf{f})$.
—**sumlist:** specifies the relation between a list of numbers and the sum of its elements; entry pattern $sumlist(\mathbf{d}, \mathbf{f})$.
—**trap:** first-order ordinary differential equation solving, using the trapezoidal method; entry pattern $solve([\mathbf{d}, \mathbf{d}], \mathbf{d}, [\mathbf{d}, \mathbf{f}])$.
—**vecmat:** performs vector and matrix operations (vector addition *vecadd*, multiplication of a matrix and a vector *matvecmul*, and matrix multiplication *matmul*); entry patterns *vecmat*1 which gives rise to $matvecmul(\mathbf{d}, \mathbf{d}, \mathbf{f})$, $vecadd(\mathbf{f}, \mathbf{d}, \mathbf{d})$, and $matmul(\mathbf{d}, \mathbf{d}, \mathbf{f})$, and *vecmat*2 which gives rise to $matvecmul(\mathbf{f}, \mathbf{d}, \mathbf{d})$, $vecadd(\mathbf{f}, \mathbf{d}, \mathbf{a})$, and $matmul(\mathbf{d}, \mathbf{f}, \mathbf{d})$.

The other CLP benchmarks are obtained from P. Van Hentenryck (*bridge*, *cutstock*, *warehouse*), from Van Hentenryck [1989, (*magic* (p. 155), *perm* (p. 152))] and Van Hentenryck and Ramachandran [1994, (*periodic* (p. 350))], from PrologIA (*color*4, *color*4*F*, *triangle*), and from Colmerauer [1990] (*magicC*).

Most of the LP benchmarks are used in Mulkers et al. [1994], from which we borrow the following brief description of the programs. *akl* (called *init_vars* in Mulkers et al. [1994]) initializes two abstract substitutions to have the same set of variables; *akl_old* is a slightly modified version of *akl*; *ann* is a simplified version of &-Prolog's parallelizing annotator [Hermenegildo and Greene 1990]; *bid* computes an opening bid for a bridge hand; *boyer* is a Boyer-Moore theorem prover from the Gabriel benchmarks (as translated by E. Tick); *browse* is a program for pattern matching also taken from the Gabriel benchmarks (as translated by T. Dobry and H. Touati); *deriv* performs symbolic differentiation of an equation; *grammar* is a program that generates and recognizes a small set of English; *icomp* is a code generator for the WAM, written by Demoen; *kalah* is the Kalah playing program from Sterling and Shapiro [1994] which uses alpha-beta pruning; *mapcolor* is a map-coloring program for a map representation of six countries; *peephole* is the optimizer of SB-Prolog, written by Debray; *rdtok* is O'Keefe's public domain Prolog tokenizer; *read* is Warren and O'Keefe's public domain Prolog parser; *serialize* is a program manipulating lists of numbers; *tarjan* is a program for computing strongly connected components written by Gallagher; *vlokgr* is a consistency checker for a lectures-administration database, written by Janssens; *vlok* is the same program but using an open-ended list for the list of lectures to be checked. The remaining benchmarks are the following: *append* is the well-known append program; *pg* is a program written by W. Older to solve a specific mathematical problem; *plan* is a simple planner in the blocks world; *qsort* implements the quicksort algorithm; *queens* is a generate-and-test program to solve the n-queens problem; *qplan* is part of CHAT, a natural language query interpreter; *witt* is a conceptual clustering system (written by Manuel Carro).

ACKNOWLEDGMENTS

The authors would like to thank Francisco Bueno for his help in improving the implementation of the analysis framework and German Puebla for modifying the system in order to output the inferred information for all the analyzed versions. Thanks are due to Pascal Van Hentenryck, PrologIA, and the other PRINCE project partners for providing the benchmark programs. The authors are also grateful to the anonymous referees for their suggestions which have improved both the contents and the presentation of the article.

REFERENCES

Armstrong, T., Marriott, K., Schachte, P., and Søndergaard, H. 1994. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In *Proceedings of the Static Analysis Symposium*, B. Le Charlier, Ed. Lecture Notes in Computer Science, vol. 864. Springer-Verlag, Namur, Belgium, 266–280.

Bruynooghe, M. 1991. A practical framework for the abstract interpretation of logic programs. *J. Logic Program. 10,* 2 (Feb.), 91–124.

Bruynooghe, M. and Boulanger, D. 1994. Abstract interpretation for (constraint) logic programming. In *Constraint Programming*, B. Mayoh, E. Tyugu, and J. Penjam, Eds. Nato ASI Series, vol. F/131. Springer-Verlag, Berlin, 228–258.

Bruynooghe, M. and Janssens, G. 1992. Propagation: A new operation in a framework for abstract interpretation of logic programs. In *Proceedings of the 3rd International Workshop on Metaprogramming in Logic*, A. Pettorossi, Ed. Lecture Notes in Computer Science, vol. 649. Springer-Verlag, Uppsala, Sweden, 294–307.

Bueno, F., de la Banda, M. G., and Hermenegildo, M. 1994. Effectiveness of global analysis in strict independence-based automatic program parallelization. In *Proceedings of the 1994 International Symposium on Logic Programming.* MIT Press, Cambridge, Mass., 320–336.

Codish, M., Mulkers, A., Bruynooghe, M., García de la Banda, M., and Hermenegildo, M. 1995. Improving abstract interpretations by combining domains. *ACM Trans. Program. Lang. Syst. 17,* 1 (Jan.), 28–44.

Codognet, P. and Filé, G. 1992. Computations, abstractions and constraints in logic programs. In *Proceedings of the 4th International Conference on Computer Languages*, J. Cordy, Ed. IEEE Computer Society Press, Los Alamitos, Calif., 155–164.

Colmerauer, A. 1990. An introduction to PROLOGIII. *Commun. ACM 30,* 7 (July), 69–96.

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

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages.* ACM, New York, 269–282.

Cousot, P. and Cousot, R. 1992a. Abstract interpretation and application to logic programs. *J. Logic Program. 13,* 2 – 3 (July), 103–179.

Cousot, P. and Cousot, R. 1992b. 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*, M. Bruynooghe and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 631. Springer-Verlag, Leuven, Belgium, 269–295.

Dart, P. 1988. Dependency analysis and query interfaces for deductive databases. Ph.D. thesis, Univ. of Melbourne, Australia.

Debray, S. K. 1989. Static inference of modes and data dependencies in logic programs. *ACM Trans. Program. Lang. Syst. 11,* 3, 418–450.

Debray, S. K. 1992a. Efficient dataflow analysis of logic programs. *J. ACM 39,* 4 (Oct.), 949–984.

Debray, S. K., Ed. 1992b. *Special issue: Abstract interpretation. J. Logic Program. 13,* 2 – 3 (July).

Debray, S. K. 1995. On the complexity of dataflow analysis of logic programs. *ACM Trans. Program. Lang. Syst. 17,* 2 (Mar.), 331–365.

Dumortier, V. 1994. Freeness and related analyses of constraint logic programs using abstract interpretation. Ph.D. thesis, Dept. of Computer Science, Katholieke Univ. Leuven, Leuven, Belgium.

Dumortier, V. and Janssens, G. 1994. Towards a practical full mode inference system for CLP(H,N). In *Proceedings of the 11th International Conference on Logic Programming*, P. Van Hentenryck, Ed. MIT Press, Cambridge, Mass., 569–583.

Dumortier, V., Janssens, G., Bruynooghe, M., and Codish, M. 1993. Freeness analysis in the presence of numerical constraints. In *Proceedings of the 10th International Conference on Logic Programming*, D. S. Warren, Ed. MIT Press, Cambridge, Mass., 100–115.

Englebert, V., Le Charlier, B., Roland, D., and Van Hentenryck, P. 1992. Generic abstract interpretation algorithms for Prolog: Two optimization techniques and their experimental evaluation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming* (*PLILP 92*), M. Bruynooghe and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 631. Springer-Verlag, Leuven, Belgium, 311–325. Also in *Software Practice and Experience*, 23(4):419–460, 1993.

García de la Banda, M. 1994. Independence, global analysis, and parallelism in dynamically scheduled constraint logic programming. Ph.D. thesis, Univ. Politécnica de Madrid, Spain.

García de la Banda, M. and Hermenegildo, M. 1993. A practical approach to the global analysis of CLP programs. In *Proceedings of the 1993 International Logic Programming Symposium*, D. Miller, Ed. MIT Press, Cambridge, Mass., 437–455.

García de la Banda, M., Hermenegildo, M., and Marriott, K. 1993. Independence in constraint logic programs. In *Proceedings of the 1993 International Logic Programming Symposium*, D. Miller, Ed. MIT Press, Cambridge, Mass., 130–146.

García de la Banda, M., Marriott, K., and Stuckey, P. 1995. Efficient analysis of logic programs with dynamic scheduling. In *Logic Programming, Proceedings of the 1995 International Symposium* (*ILPS'95*), J. LLoyd, Ed. MIT Press, Cambridge, Mass., 417–431.

Giacobazzi, R., Debray, S., and Levi, G. 1993. Generalized semantics and abstract interpretation for constraint logic programs. Draft, Univ. of Pisa. Apr. Preliminary version in Proceedings of the International Conference on Fifth Generation Computer Systems 1992.

Hanus, M. 1993. Analysis of nonlinear constraints in CLP(R). In *Proceedings of the 10th International Conference on Logic Programming*, D. S. Warren, Ed. MIT Press, Cambridge, Mass., 83–99.

Hanus, M. 1995. Analysis of residuation in logic programs. *J. Logic Program. 24,* 3 (Sept.), 161–199.

HERMENEGILDO, M. AND GREENE, K. J. 1990. &-Prolog and its performance: Exploiting independent And-parallellism. In *Proceedings of the 7th International Conference on Logic Programming*, D. H. D. Warren and P. Szeredi, Eds. MIT Press, Cambridge, Mass., 253–268.

HERMENEGILDO, M., MARRIOTT, K., PUEBLA, G., AND STUCKEY, P. 1995. Incremental analysis of logic programs. In *Proceedings of the 12th International Conference on Logic Programming*, L. Sterling, Ed. MIT Press, Cambridge, Mass., 797–811.

JACOBS, D. AND LANGEN, A. 1992. Static analysis of logic programs for independent And-parallelism. *J. Logic Program. 13,* 2 – 3 (July), 291–314.

JAFFAR, J. AND LASSEZ, J.-L. 1987. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on the Principles of Programming Languages.* ACM, New York, 111–119.

JAFFAR, J. AND MAHER, M. 1994. Constraint logic programming: A survey. *J. Logic Program. 19 – 20*, 503–581.

JAFFAR, J., MICHAYLOV, S., STUCKEY, P., AND YAP, R. 1992. An abstract machine for CLP($\mathcal{R}$). In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, 128–139.

JANSSENS, G. AND BRUYNOOGHE, M. 1992. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic Program. 13,* 2 – 3 (July), 205–258.

JANSSENS, G., BRUYNOOGHE, M., AND DUMORTIER, V. 1995. A blueprint for an abstract machine for abstract interpretation of (constraint) logic programs. In *Logic Programming, Proceedings of the 1995 International Symposium* (*ILPS'95*), J. LLoyd, Ed. MIT Press, Cambridge, Mass., 336–350.

JØRGENSEN, N., MARRIOTT, K., AND MICHAYLOV, S. 1991. Some global compile-time optimizations for CLP($\mathcal{R}$). In *Proceedings of the 1991 International Symposium on Logic Programming*, V. Saraswat and K. Ueda, Eds. MIT Press, Cambridge, Mass., 420–434.

LASSEZ, J.-L. AND MCALOON, K. 1992. A canonical form for generalised linear constraints. *J. Symb. Comput. 13,* 1 (Jan.), 1–24.

LE CHARLIER, B. AND VAN HENTENRYCK, P. 1994. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Trans. Program. Lang. Syst. 16,* 1 (Jan.), 35–101.

LE CHARLIER, B., MUSUMBU, K., AND VAN HENTENRYCK, P. 1991. A generic abstract interpretation algorithm and its complexity analysis (extended abstract). In *Proceedings of the 8th International Conference on Logic Programming*, K. Furukawa, Ed. MIT Press, Cambridge, Mass., 64–78.

LE CHARLIER, B., ROSSI, S., AND VAN HENTENRYCK, P. 1994. An abstract interpretation framework for almost full prolog. In *Proceedings of the 1994 International Logic Programming Symposium*, M. Bruynooghe, Ed. MIT Press, Cambridge, Mass.

LLOYD, J. W. 1987. *Foundations of Logic Programming*, 2nd ed. Symbolic Computation — Artificial Intelligence. Springer-Verlag, Berlin.

MACDONALD, A. D., STUCKEY, P. J., AND YAP, R. H. C. 1993. Redundancy of variables in CLP($\mathcal{R}$). In *Proceedings of the 1993 International Logic Programming Symposium*, D. Miller, Ed. MIT Press, Cambridge, Mass., 75–93.

MARRIOTT, K. 1993. Frameworks for abstract interpretation. *Acta Inf. 30*, 103–129.

MARRIOTT, K. AND SØNDERGAARD, H. 1989. Semantics-based dataflow analysis of logic programs. In *Information Processing 89*, G. Ritter, Ed. Elsevier Science Publishers B.V., North-Holland, Amsterdam, 601–606.

MARRIOTT, K. AND SØNDERGAARD, H. 1990. Analysis of constraint logic programs. In *Proceedings of the 1990 North American Conference on Logic Programming*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, Mass., 531–547.

MARRIOTT, K. AND STUCKEY, P. 1993. 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.* ACM, New York, 334–344.

MARRIOTT, K. AND STUCKEY, P. 1994. Approximating interaction between linear arithmetic constraints. In *Proceedings of the 1994 International Symposium on Logic Programming*, M. Bruynooghe, Ed. MIT Press, Cambridge, Mass., 571–585.

MARRIOTT, K., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1994. Analyzing logic programs with dynamic scheduling. In *Proceedings of the 20th Annual ACM Conference on Principles of Programming Languages*. ACM, New York, 240–253.

MARRIOTT, K., SØNDERGAARD, H., STUCKEY, P., AND YAP, R. 1994. Optimizing compilation for CLP($\mathcal{R}$). In *Proceedings of the 17th Annual Computer Science Conference.*

MARTELLI, A. AND MONTANARI, U. 1982. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4,* 3, 258–282.

MELLISH, C. 1986. Abstract interpretation of Prolog programs. In *Proceedings of the 3rd International Conference on Logic Programming*, E. Shapiro, Ed. Lecture Notes in Computer Science, vol. 225. Springer-Verlag, Berlin, 463–475.

MULKERS, A. 1993. *Live Data Structures in Logic Programs, Derivation by Means of Abstract Interpretation.* Lecture Notes in Computer Science, vol. 675. Springer-Verlag, Berlin.

MULKERS, A., SIMOENS, W., JANSSENS, G., AND BRUYNOOGHE, M. 1994. On the practicality of abstract equation systems. Tech. Rep. CW198, Dept. of Computer Science, Katholieke Univ. Leuven, Leuven, Belgium. Nov.

MULKERS, A., SIMOENS, W., JANSSENS, G., AND BRUYNOOGHE, M. 1995. On the practicality of abstract equation systems. In *Proceedings of the 12th International Conference on Logic Programming*, L. Sterling, Ed. MIT Press, Cambridge, Mass., 781–795.

MULKERS, A., WINSBOROUGH, W., AND BRUYNOOGHE, M. 1990. Analysis of shared data structures for compile-time garbage collection in logic programs. In *Proceedings of the 7th International Conference on Logic Programming*, D. H. D. Warren and P. Szeredi, Eds. MIT Press, Cambridge, Mass., 747–762.

MULKERS, A., WINSBOROUGH, W., AND BRUYNOOGHE, M. 1994. Live-structure dataflow analysis for Prolog. *ACM Trans. Program. Lang. Syst. 16,* 2 (Mar.), 205–258.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1989. Determination of variable dependence information at compile-time through abstract interpretation. In *Proceedings of the 1989 North American Conference on Logic Programming*, E. Lusk and R. Overbeek, Eds. MIT Press, Cambridge, Mass., 166–189.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1990. Deriving a fixpoint computation algorithm for top-down abstract interpretation of logic programs. Tech. Rep. ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, Tex. Apr.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1992. Compile-time derivation of variable dependency using abstract interpretation. *J. Logic Program. 13,* 2 – 3 (July), 315–347.

NIELSON, F. 1988. Strictness analysis and denotational abstract interpretation. *Inf. Comput. 76,* 1, 29–92.

PLAISTED, D. A. 1984. The occur-check problem in Prolog. *New Gen. Comput. 2,* 4, 309–322. Also in *Proceedings of the 1984 International Symposium on Logic Programming.*

RAMACHANDRAN, V. AND VAN HENTENRYCK, P. 1995. LSign reordered. In *International Static Analysis Symposium* (*SAS'95*). Lecture Notes in Computer Science, vol. 983. Springer-Verlag, Berlin, 330–347.

STERLING, L. AND SHAPIRO, E. 1994. *The Art of Prolog: Advanced Programming Techniques*, 2nd ed. Logic Programming Series. MIT Press, Cambridge, Mass.

VAN HENTENRYCK, P. 1989. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Mass.

VAN HENTENRYCK, P. AND RAMACHANDRAN, V. 1994. Backtracking without trailing in CLP($\mathcal{R}_{Lin}$). In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, 349–360.

VAN ROY, P. AND DESPAIN, A. M. 1992. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Comput. 25,* 1 (Jan.), 54–68.

WARREN, R., HERMENEGILDO, M., AND DEBRAY, S. 1988. On the practicality of global flow analysis of logic programs. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, R. Kowalski and K.A.Bowen, Eds. MIT Press, Cambridge, Mass., 684–699.

Received January 1995; revised October 1995 and February 1996; accepted April 1996