A Generic Algorithm for CLP Analysis

Nevin Heintze
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Joxan Jaffar
IBM Thomas J. Watson Research Center
PO Box 704, Yorktown Heights, NY 10598

Abstract

Logic program analyzers typically employ abstract interpretation and consist of two main components: an abstract domain and a generic iterative fixed-point computation or "engine". In earlier work, we presented a new engine based on an unfolding operation, and showed that it was uniformly more accurate than the standard engine. This unfolding engine however had a crucial limitation: it could not be generalized to constraint logic programming (CLP).

In this paper, we develop a fundamentally different view of unfolding, based on a uniform treatment of constraints. The result is a family of analysis algorithms which is modular in the sense that CLP is modular over computation domains. Not only is the new engine more general, but it is also simpler and more flexible. In particular, it provides new opportunities for tuning efficiency/accuracy tradeoffs. At one end of the spectrum, the engine can be used to describe the standard engine; at the other end is an algorithm that is more accurate than the old unfolding engine for traditional logic programs. The main technical result however is that the new engine is uniformly more accurate than a comparable abstract interpretation algorithm for CLP, whatever the constraint and abstract domains used.

1 Introduction

Most algorithms for logic program (LP) analysis are based on abstract interpretation, and consist of two components: an abstract domain (and associated functions) and an iterative fixed-point computation. Substantive differences between algorithms typically appear in the abstract domain, and design of this component that is the main focus of algorithm development (in fact it is the choice of this component that largely determines both the accuracy and cost of the analysis). In contrast, differences in the fixed-point components are relatively minor, and it is possible to speak of an idealized generic algorithm or "engine" that encompasses the essence of the fixed-point computation. Some specific descriptions of engines are [9], for logic programs, and more recently, [10], for constraint logic programs.

In earlier work [6], a new engine, called the *unfolding* engine, was presented for the analysis of logic programs. Its main advantage was improved accuracy over the standard engine, regardless of the abstract domain used. A fundamental difference between the unfolding and standard engines is that the standard engine iterates over a fixed collection of semantic equations, whereas the unfolding engine iterates over dynamically changing equations. Roughly speaking, the unfolding engine combines set constraint techniques [4, 5, 3] with abstract interpretation techniques [1]. The former provides a superior ability to reason about term structures, but ignores all dependencies between variables; the latter provides an ability to reason about dependencies. Unfortunately, the unfolding engine relies directly on a specialized representation using substitutions, which cannot be lifted to constraint logic programming (CLP). This is a pity since analysis for CLP is arguably more important than for LP – the opportunities for compiler optimizations are considerably greater (see eg. [11]).

This paper presents an unfolding engine for CLP. The main technical difference from the previous engine is a novel use of input/output variables in the constraints, as opposed to the manipulation of substitutions. What results is not just a more general engine, but a far simpler and more accurate one. Specifically:

- It provides an elegant, generic algorithm based on the well-understood primitive operation of unfolding.
- It is provides new opportunities for tuning efficiency/accuracy tradeoffs. Not only can one consider modifying the abstract domain, but various aspects of the engine itself can be controlled. At one end of the spectrum, the engine can be used to describe the standard engine; at the other end is an algorithm that provides significantly richer information that in general cannot be computed using the standard engine (no matter what abstract domain is used).
- In the special case of traditional logic programs, it is simpler, more efficient and uniformly more accurate than the old engine.
- It is uniformly more accurate than the standard engine for CLP, whatever abstract and constraint domains are used. This in fact represents the main technical result of this paper.

We next illustrate some differences between a standard abstract interpretation algorithm, and the old and new engines. Consider the abstract domain *LSign* [11, 12] which abstracts the coefficients in linear arithmetic expressions by their sign. Thus for example, $3X - 4Y = 5$ is described by the *LSign* formula $\oplus X \ominus Y = \oplus$ where $\oplus$ denotes some positive number, and $\ominus$ some negative number.

```
?- p(X,Y), q(X).
p(A,B) :- A - 2*B = 3.
q(A) :- A = 5.
```

```
?- p(f(Y,3)).
p(U) :- q(U).
q(f(2+2,V)).
```

```
?- p(X), r(X,Y).
p(U) :- q(U).
q(f(A,B)) :-
    A - 2*B = 3.
r(f(A,B),B) :-
    A + 2*B = 7.
```

For the first CLP($\mathcal{R}$) [7] program, the standard algorithm using this domain would be able to infer that Y is ground (because it can combine the formula $\oplus X \ominus Y = \oplus$ obtained from the first rule with the formula $\oplus X = \oplus$ obtained from the second rule). It cannot however infer the same for Y in the second program, which requires a completely different kind of reasoning, that over partially instantiated term structures. Here a set constraint approach would have been useful, if only it could handle arithmetic expressions and constraints. The third program serves as a combination of the first two, and exemplifies where the new engine, which will infer the *exact* value for Y, retains the power of all previous engines, and more.

2 Preliminary Definitions

Let $\mathcal{V}$, Σ, Π_p and Π_c be countably infinite collections of variables, function symbols, *program predicates* and *constraint predicates* respectively. Each function and predicate symbol has a given arity. A term is either a variable or of the form $f(t_1, \ldots, t_n)$ where $f \in \Sigma$ has arity n and the t_i are terms. An *atom* is of the form $p(t_1, \ldots, t_n)$ where $p \in \Pi_p$ has arity $n \geq 0$ and each t_i is a term. An *atomic constraint* is an expression of the form $p(t_1, \ldots, t_n)$ where $p \in \Pi_c$ has arity $n \geq 0$ and each t_i is a term. A *program* is a collection of rules of the form $H \leftarrow B_1, \ldots, B_m$, where $m \geq 0$, H is an atom, and each B_i is either an atom or an atomic constraint. We refer to H as the *head* of the rule, and to the sequence $B_1, \ldots, B_m$ as the *body* of the rule.

A *constraint* is either an atomic constraint or of one of the forms $(\exists X)(c)$ or $c_1 \wedge \cdots \wedge c_n$ where $c, c_1, \ldots, c_n$ are constraints, and X is a program variable. Bound and free occurrences of variables in a constraint are defined in the usual manner. Let $var(e)$ denote the set of free variables in the expression e. Then if $S = \{X_1, \ldots, X_n\}$ is a set of variables, we write $(\exists_S)(c)$ to denote $(\exists X_1) \cdots (\exists X_n)(c)$, and we write $(\exists_{\overline{S}})(c)$ to denote $(\exists_T)(c)$ where $T = var(c) - S$.

A *substitution* θ is a mapping from variables into terms. Where e is some expression, $e\theta$ denotes the simultaneous replacement of each free occurrence of a variable X in e by $X\theta$ (bound variables are appropriately renamed to avoid "variable capture", so that if θ maps X into Y, then $(\exists Y)(X = f(Y))$ is mapped to $(\exists Y')(Y = f(Y'))$ under θ). A *renaming* substitution θ is an injection from $\mathcal{V}$ into $\mathcal{V}$; we say that θ *renames away* from an expression e if the range of θ is disjoint from $var(e)$.

The meaning of constraints is given by a *structure* $\mathcal{J}$ that defines the underlying domain of computation, $\mathcal{X}$, assigns a meaning $\mathcal{J}(f) : \mathcal{X}^n \rightarrow \mathcal{X}$ to each n-ary function symbol in $f \in \Sigma$, and assigns a meaning $\mathcal{J}(p) : \mathcal{X}^n \rightarrow \{true, false\}$ to each n-ary constraint predicate in $p \in \Pi_c$. $\mathcal{X}$ is also referred to as the *constraint domain* of the CLP language. Given a valuation ρ, which maps variables into $\mathcal{X}$, and given a structure $\mathcal{J}$, the meanings of terms and constraints are defined in the obvious inductive manner. If for some valuation ρ, the meaning of a constraint c is *true*, then we say that c is *satisfiable*.

We define a bottom-up operational semantics of a program P as follows. An atom A *succeeds with answer constraint* c from P if c is satisfiable and there exists some rule $H \leftarrow B_1, \ldots, B_n$ in P such that c is $(\exists_{\overline{var(A)}})(A = H\theta \wedge c_1\theta \wedge \cdots \wedge c_n\theta)$ where θ renames away from A, and for each $1 \leq i \leq n$, either B_i is an atom that succeeds with answer constraint c_i, or else B_i is the atomic constraint c_i. Note that the free variables in the answer constraint c are exactly the variables in A.

A central concept from abstract interpretation of CLP is the notion of an abstract domain for representing constraints.

Definition 1 An abstract domain is a set of *abstract formulas* together with (a) an *abstraction function* ABS which takes a set of constraints and returns an abstract formula, and (b) a *concretization function* CONC which takes an abstract formula and returns a set of constraints satisfying $\text{CONC}(\text{ABS}(C)) \supseteq C$ for constraint sets C. (Typically, some further algebraic conditions are specified, but these will not concern us here.) $\square$

Using such domains, the Appendix presents an idealized abstract interpretation engine for CLP that encompasses the essence of the iterative fixed-point algorithms used in the literature.

3 Semantic Equations

We now give an equational formulation of the bottom-up semantics defined in the previous section. Whereas the previous definition of the operational semantics focuses on answer constraints generated for a specific atom, the following equational definition focuses on the constraints generated from the body of a rule. To this end, let the rules of program P be $R_1, \ldots, R_N$, and let H_i denote the head of the rule R_i. Now, for each $1 \leq i \leq N$, let c denote the conjunction of the atomic constraints appearing in the body of R_i, let $A_1, \ldots, A_m$ denote the subsequence of atoms appearing in the body of R_i, and construct the equation

$$\Psi_i = \bigcup_{j_1,\ldots,j_m} \left\{ c \wedge \left(\begin{array}{c} \exists_{S_1} (B_1 = H_{j_1}\theta \ \wedge \ c_1\theta) \\ \wedge \ \cdots \ \wedge \\ \exists_{S_n} (B_m = H_{j_m}\theta \ \wedge \ c_m\theta) \end{array} \right) \ : \ c_1 \in \Psi_{j_1}, \ldots, c_m \in \Psi_{j_m} \right\}$$

where $j_1, \ldots, j_m$ ranges over $m - tuples$ such that $H_{j_1}, \ldots, H_{j_m}$ (after being suitably renamed) unifies with $B_1, \ldots, B_m$, and θ renames away from $B_1, \ldots, B_m$. This formulation of semantic equations represents, for each rule R_i, the set of collected constraints on the variables of R_i obtained from executing the body of R_i. Intuitively, a semantic equation $\Psi_i = \cdots$ provides a definition for the variables in the rule R_i. A *solution* of such equations is a mapping $\mathcal{I}$ from each Ψ_i into a set of constraints whose free variables are from $var(R_i)$, such that each equation is satisfied[1]. For example, consider the CLP($\mathcal{R}$) program:

```
1.   fac(0, 1).
2.   fac(1, 1).
3.   fac(N, F) :- N > 1, F = N * F1, fac(N - 1, F1).
```

After introducing variables Ψ_1, Ψ_2 and Ψ_3 corresponding to rules 1, 2 and 3 respectively, we have the semantic equations

$$\begin{aligned}
\Psi_1 &= \{true\} \\
\Psi_2 &= \{true\} \\
&\quad \{N > 1 \wedge F = N * F1 \wedge fac(N-1, F1) = fac(0,1) \wedge c\theta : c \in \Psi_1\} \ \bigcup \\
\Psi_3 &= \{N > 1 \wedge F = N * F1 \wedge fac(N-1, F1) = fac(1,1) \wedge c\theta : c \in \Psi_2\} \ \bigcup \\
&\quad \{N > 1 \wedge F = N * F1 \wedge \exists N' \exists F' \, (fac(N-1, F1) = fac(N', F') \wedge c\theta) : c \in \Psi_3\}
\end{aligned}$$

where *true* denotes the empty conjunction of constraints, and the new variables N' and S' are renamings of N and S by θ. The following is a solution of these constraints:

$$\begin{array}{l} \Psi_1 = \{true\} \\ \Psi_2 = \{true\} \end{array} \qquad \Psi_3 = \left\{ \left(\begin{array}{c} N = 2 \\ F = 2 \\ F1 = 1 \end{array} \right), \left(\begin{array}{c} N = 3 \\ F = 6 \\ F1 = 2 \end{array} \right), \left(\begin{array}{c} N = 4 \\ F = 24 \\ F1 = 6 \end{array} \right), \ldots \right\}$$

This is in fact the least such solution (solutions are ordered component-wise: $\mathcal{I}_1 \leq \mathcal{I}_2$ iff $\mathcal{I}_1(\Psi_i) \subseteq \mathcal{I}_2(\Psi_i)$ for each Ψ_i), and it coincides exactly with the program's meaning. That is, Ψ_i consists of just the answer constraints obtained when the body of the i^{th} rule is executed as a goal. We conclude this section with a formal statement connecting the original operational semantics with the equational formulation.

[1]We tacitly identify constraints that are semantically equivalent (i.e. are satisfied by the same valuations). This issue is discussed in section 6.

Proposition 1 *Let $\mathcal{SE}_P$ be the semantic equations for a program P, and let $\mathcal{I}$ be the least model of $\mathcal{SE}_P$. Then, an atom A succeeds with answer constraint c iff, for some i, there is a constraint $c_i \in \mathcal{I}(\Psi_i)$ such that c is $(\exists_{\overline{var(A)}}(A = H_i\theta \wedge c_i\theta))$ where θ renames away from A.*

4 Unfolding Semantic Equations

At the heart of our algorithm is the notion of unfolding semantic equations by substituting one semantic equation into another. In essence, the individual expressions in semantic equations define fragments of constraints, which are then combined to give the semantics of the entire program. To see this, recall that each semantic equation has the form $\Psi_i = exp_1 \cup \cdots \cup exp_r$ where each exp_l has the form

$$\left\{ c \wedge \left(\begin{array}{c} \exists_{\overline{var(B_1)}}(B_1 = H_{j_1}\theta \;\wedge\; c_1\theta) \\ \wedge\; \cdots \;\wedge \\ \exists_{\overline{var(B_m)}}(B_m = H_{j_m}\theta \;\wedge\; c_m\theta) \end{array} \right) : \; c_1 \in \Psi_{j_1}, \ldots, c_m \in \Psi_{j_m} \right\} \qquad (4.1)$$

Now, consider the expressions of the form

$$\exists_{\overline{var(B_k)}}(B_k = H_{j_k}\theta \;\wedge\; c_k\theta), \;\; 1 \leq k \leq m \qquad (4.2)$$

appearing in each exp_l. Recall that Ψ_i is interpreted as a set of constraints whose free variables are from $var(R_i)$. Hence c_k is a constraint whose free variables are from $var(R_{j_k})$. Moreover, all of the free variables in $H_{j_k}\theta$ and $c_k\theta$ will be bound by $\exists_{\overline{var(B_k)}}$. One can therefore view (4.2) as a function that maps a constraint c_k with free variables from $var(R_{j_k})$ into a constraint with free variables from $var(R_i)$.

The central component of this function is the pair of terms B_k and H_{j_k}, since these two terms completely determine the connection between the variables in $var(R_i)$ and $var(R_{j_k})$. In effect H_{j_k} matches with the variables in c_k, and the "constraint fragment" $B_k = H_{j_k}$ can be thought of as mapping c_k into a constraint on $var(R_i)$. Intuitively, we would like to write $B_k = H_{j_k}$ as a constraint and consider conjoining it with c_k, but we cannot do so because the variables in H_{j_k} must be renamed (θ appears in (4.1) for precisely this reason). At a conceptual level, we can avoid this issue by viewing the variables B_k and H_{j_k} as coming from two disjoint name spaces – that is, view the variables in B_k as "output" variables and those in H_{j_k} as "input" variables.

To formalize this, consider augmenting the universe of variables $\mathcal{V}$ by two parallel universes : the input variables $\mathcal{V}\!\uparrow$ and the output variables $\mathcal{V}\!\downarrow$. Both $\mathcal{V}\!\uparrow$ and $\mathcal{V}\!\downarrow$ are assumed to be countably infinite and disjoint from each other and from $\mathcal{V}$. We shall view these new sets of variables as "copies" (or renamings) of $\mathcal{V}$. For each variable $X \in \mathcal{V}$, let $X^{\uparrow}$ and $X^{\downarrow}$ denote the corresponding variables in $\mathcal{V}\!\uparrow$ and $\mathcal{V}\!\downarrow$. Define a distinguished substitution $\uparrow$ as follows: $(X^{\downarrow})\!\uparrow = X$, $(X)\!\uparrow = X^{\uparrow}$, $(X^{\uparrow})\!\uparrow = X^{\uparrow}$. In other words, $\uparrow$ is a "shift up" operation that maps $\mathcal{V}^{\downarrow}$ into $\mathcal{V}$, and maps $\mathcal{V}$ into $\mathcal{V}^{\uparrow}$, and is the identity on $\mathcal{V}^{\uparrow}$. Similarly, we have a distinguished substitution $\downarrow$ as follows: $(X^{\uparrow})\!\downarrow = X$, $(X)\!\downarrow = X^{\downarrow}$, $(X^{\downarrow})\!\downarrow = X^{\downarrow}$.

Using these definitions, we can write the constraint $B_k\!\uparrow = H_{j_k}\!\downarrow$ to represent the connection between B_k and H_{j_k}. That is, B_k is written using output variables, and H_{j_k} uses input variables. Note that $B_k\!\uparrow = H_{j_k}\!\downarrow$ can be treated as a standard constraint (for example, we can apply usual constraint simplification procedures to it). Formulas of the form (4.1) can now be written concisely as

$$\left(\begin{array}{c} [c\uparrow]_\star \\ [B_1\uparrow = H_{j_1}\downarrow]_{j_1} \\ \cdots \\ [B_m\uparrow = H_{j_m}\downarrow]_{j_m} \end{array}\right)$$

We call such expressions *clusters*. In general, a cluster has the following form[2]

$$\left(\begin{array}{c} [c_0]_\star \\ [c_1]_{j_1} \\ \cdots \\ [c_n]_{j_n} \end{array}\right) \tag{4.3}$$

where each c_k, $0 \le k \le n$, is a constraint, subject to the following invariant: if the cluster appears on the right hand side of the equation for Ψ_i, then

- the free variables in c_0 are from $R_i\uparrow$, and
- the free variables in each c_k, $k \ge 1$, are from $R_i\uparrow$ and $R_{j_k}\downarrow$.

Each expression $[c_k]_{j_k}$ is called a *group*, and j_k is called the *dependency* of the group (indicating which Ψ_i variable the group depends on). The group $[c_0]_\star$ is said to be a *terminal group* since it does not depend on any Ψ_i. Given a function $\mathcal{I}$ that maps from each Ψ_i to a set of constraints whose free variables are from $var(R_i)$, the cluster (4.3) is interpreted as the following collection of constraints:

$$\left\{ \left(\begin{array}{c} c_0 \wedge \\ (\exists_{S_1})(c_1\uparrow \;\wedge\; c'_1) \\ (\exists_{S_n})(c_n\uparrow \;\wedge\; c'_n) \end{array}\right)\Big\downarrow : \; c'_1 \in \mathcal{I}(\Psi_{j_1}), \ldots, c'_n \in \mathcal{I}(\Psi_{j_n}) \right\}$$

where each S_k is $var(R_{j_k})$. To explain this definition, suppose that this cluster appears on the right hand side of the equation for Ψ_i. Then each c_k, $k \ge 1$, is a constraint whose free variables are either input variables from $var(R_{j_k}\downarrow)$ or output variables from $var(R_i\uparrow)$, and each c'_k will be a constraint whose free variables come from $var(R_{j_k})$. Hence, in $(c_k\uparrow \;\wedge\; c'_k)$, the input variables of c_k are mapped to the variables $var(R_{j_k})$, matching up with those in c'_k. The output variables in c_k are unaffected by the $\uparrow$ operation. So, each expression $(\exists_{S_k})(c_k\uparrow \;\wedge\; c'_k)$ denotes a constraint whose free variables are from $var(R_i\uparrow)$. Finally, a $\downarrow$ operation is applied to the conjunction of c_0 and these n constraints, and the result is a constraint whose free variables are from $var(R_i)$.

For example, when writing the equations for the factorial program in this new format (and doing some obvious constraint simplification), we obtain:

$$\begin{array}{lcl} \Psi_1 & = & () \\ \Psi_2 & = & () \\ \Psi_3 & = & ([\mathit{false}]_1) \;\bigcup\; \left(\left[\begin{array}{c} N^\uparrow = 2 \\ F^\uparrow = 2 \\ F1^\uparrow = 1 \end{array}\right]_2\right) \;\bigcup\; \left(\left[\begin{array}{c} N^\uparrow > 1 \\ F^\uparrow = N^\uparrow * F1^\uparrow \\ N^\uparrow - 1 = N^\downarrow \\ F1^\uparrow = F^\downarrow \end{array}\right]_3\right) \end{array}$$

Consider below another example program and its semantic equations, illustrating

[2] If c_0 is the empty conjunction *true*, then we sometimes omit it.

constraints and clusters of more than one group.

```
1.    p(X, Y, Z) :- X - Y = 1,
                    q(X, Y),
                    r(Z).
2.    q(U, V) :- U + V = 3.
3.    r(42).
```

$$\Psi_1 = \left(\begin{array}{c} [X^\uparrow - Y^\uparrow = 1]_\star \\ \left[\begin{array}{c} X^\uparrow = U^\downarrow \\ Y^\uparrow = V^\downarrow \end{array} \right]_2 \\ [Z^\uparrow = 42]_3 \end{array} \right)$$

$$\Psi_2 = ([U^\uparrow + V^\uparrow = 3]_\star)$$

$$\Psi_3 = ()$$

Note that the semantic equations we have defined correspond to a bottom-up execution of the program P at hand. We can also define semantic equations corresponding to a top-down execution of P. In what follows, the term semantic equations shall mean either the equations corresponding to bottom-up or top-down program execution. Our engine can be applied in either case.

We now describe the basic step of our engine: the unfolding of semantic equations. Consider a cluster containing only groups with null dependencies. Such a cluster has a fixed interpretation (it does not depend on the assignment to any Ψ_i variable). Call such clusters $\mathcal{G}$ *terminal*. For example, for the semantic equations corresponding to the program above, the clusters appearing in the equations for Ψ_2 and Ψ_3 are both terminal. Now, let C_1, C_2 and C_3 denote the clusters on the right hand sides of the equations for Ψ_1, Ψ_2 and Ψ_3 respectively, and consider substituting the terminal cluster C_2 into the first group of C_1 (since this group depends on Ψ_2). The resulting cluster (after some constraint simplification) is

$$\left(\begin{array}{c} [X^\uparrow - Y^\uparrow = 1]_\star \\ [X^\uparrow + Y^\uparrow = 3]_\star \\ [Z^\uparrow = 42]_3 \end{array} \right)$$

which can be further simplified into $([X^\uparrow = 2 \wedge Y^\uparrow = 1]_\star \;\; [Z^\uparrow = 42]_3)$. Call this resulting cluster C_4. Since C_4 is just a more "explicit" version of information already present in the semantic equations, we can add it to the equation Ψ_1 to obtain

$$\Psi_1 = \left(\begin{array}{c} [X^\uparrow - Y^\uparrow = 1]_\star \\ \left[\begin{array}{c} X^\uparrow = U^\downarrow \\ Y^\uparrow = Y^\downarrow \end{array} \right]_2 \\ [Z^\uparrow = 42]_3 \end{array} \right) \cup \left(\begin{array}{c} \left[\begin{array}{c} X^\uparrow = 2 \\ Y^\uparrow = 1 \end{array} \right]_\star \\ [Z^\uparrow = 42]_3 \end{array} \right)$$

and still preserve the meaning of the constraints. We could similarly substitute C_3 into the resulting C_4 to obtain the new cluster $([X^\uparrow = 2 \wedge Y^\uparrow = 1 \wedge Z^\uparrow = 42]_\star)$, which could again be added to the equation for Ψ_1. Note that this last cluster is a terminal cluster: in principal it could therefore be used to substitute into clusters that depend on Ψ_1 (although there are no such clusters in this example).

In summary, the unfolding step operates on (a) a cluster that contains an occurrence of a group with dependency i (hence the meaning of the cluster depends on the value of Ψ_i) and (b) a terminal cluster that appears in the right hand side of the equation for Ψ_i. The unfolding step substitutes the terminal cluster for Ψ_i in the group with dependency i. This results in a new cluster, which is added to the semantic equations. The effect of the unfolding is to collect and combine constraint fragments from various parts of the program in a way that makes information in the semantic equations more explicit. That is, after an unfolding step, more of the least solution of the semantic equations is represented in terminal clusters.

Intuitively, if we could exhaustively apply this unfolding step, then eventually all the information about the least solution would be "explicit" in the terminal clusters (in fact we could delete all non-terminal clusters without changing the least solution). However, unfolding may not terminate. For example, consider the semantic equations for the factorial program. Here unfolding produces the following sequence of terminal groups:

$$\left(\begin{bmatrix} N^{\uparrow}=2 \\ F^{\uparrow}=2 \\ F1^{\uparrow}=1 \end{bmatrix}_{\star}\right), \left(\begin{bmatrix} N^{\uparrow}=3 \\ F^{\uparrow}=6 \\ F1^{\uparrow}=2 \end{bmatrix}_{\star}\right), \left(\begin{bmatrix} N^{\uparrow}=4 \\ F^{\uparrow}=24 \\ F1^{\uparrow}=6 \end{bmatrix}_{\star}\right), \left(\begin{bmatrix} N^{\uparrow}=5 \\ F^{\uparrow}=120 \\ F1^{\uparrow}=24 \end{bmatrix}_{\star}\right), \cdots$$

To obtain a terminating algorithm from this unfolding process, we employ an abstract domain. In effect, we check to see if the size of groups grows during unfolding. If so, we approximate groups by abstract formulas, in addition to maintaining partial information about the (exact) constraints. We give the details in the next section.

5 The Engine

We first extend the definition of clusters given in the previous section to include approximate groups. An *approximate group* is defined wrt a given abstract domain $\mathcal{D}$; it is simply of the form $[\mathcal{A}]$ where $\mathcal{A} \in \mathcal{D}$. An *exact group* is of the form $[c]_\star$ or $[c]_k$ where c is a constraint, and the dependency k is the index of some program rule R_k. Some exact groups may be marked *residual* during execution of the engine. A *group* is an exact or approximate group. A *cluster* is of the form $(\mathcal{G}_1, \ldots, \mathcal{G}_n)$ where each $\mathcal{G}_i$ is a group. Our algorithm starts with the semantic equations of a program P and then modifies these equations using unfolding steps. At all times, corresponding to each rule R_k, there is one semantic equation of the form $\Psi_k = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n$ where the $\mathcal{C}_i$ are clusters. Moreover, the following invariants on the free variables of the groups in $\mathcal{C}_i$ shall be maintained:

- if $[\mathcal{A}]$ appears in $\mathcal{C}_i$ then $var(\textsc{conc}(\mathcal{A})) \subseteq var(R_k)$
- if $[c]_j$ appears in $\mathcal{C}_i$ then $var(c) \subseteq var(R_k\uparrow) \cup var(R_j\downarrow)$
- if $[c]_\star$ appears in $\mathcal{C}_i$ then $var(c) \subseteq var(R_k\uparrow)$

Given a function $\mathcal{I}$ that maps from each Ψ_i to a set of constraints whose free variables are from $var(R_i)$, each group and cluster is interpreted as a collection of constraints:

- $\mathcal{I}([\mathcal{A}]) \stackrel{\text{def}}{=} \textsc{conc}(\mathcal{A})$,
- $\mathcal{I}([c]_\star) \stackrel{\text{def}}{=} \{c\downarrow\}$,
- $\mathcal{I}([c]_k) \stackrel{\text{def}}{=} \{((\exists_{var(R_k)})(c\uparrow \ \wedge \ c'))\downarrow : c' \in \mathcal{I}(\Psi_k)\}$, and
- $\mathcal{I}\Big((\mathcal{G}_1, \ldots, \mathcal{G}_n)\Big) \stackrel{\text{def}}{=} \{c_1 \wedge \cdots \wedge c_n : c_i \in \mathcal{I}(\mathcal{G}_i)\}$.

The basic operation of our algorithm involves the substitution one cluster into another. This is defined via substitution at the level of groups. First define the *composition* of an exact group with an approximate group as follows:

$$[c]_i \circ [\mathcal{A}] \quad \stackrel{\text{def}}{=} \quad \textsc{abs}\left(\{((\exists_{var(R_i)})(c\uparrow \wedge c'))\downarrow : c' \in \textsc{conc}(\mathcal{A})\}\right)$$

Next, define the composition of an exact group $[c_1]_i$ with an exact group $[c_2]_j$ as follows[3]

$$[c_1]_i \circ [c_2]_j \stackrel{\text{def}}{=} [(\exists_{var(R_i)})(c_1{\uparrow} \wedge c_2{\downarrow})]_j$$

Now, to embed a notion of approximation into the composition $[c_1]_i \circ [c_2]_j$ of groups, define the "approximate composition" $[c_1]_i \diamond [c_2]_j$ as follows. First, compute $[c_1]_i \circ [c_2]_j$ and let the result be $[c]_j$. Then, $[c_1]_i \diamond [c_2]_j$ is defined to consist of two groups. The first group is simply the $\mathcal{D}$-approximation of $[c_1]_i \circ [c_2]_j$, that is, $[\text{ABS}(c)]$.

The second group is $[select_{c_1,c_2}(c)]_j$ where $select_{c_1,c_2}$ returns a constraint. The purpose of *select* is to maintain some of the information in $[c_1]_i \circ [c_2]_j$ while preventing unbounded growth in the semantic equations. At this point, we do not require an exact definition of *select*; we just need that it returns a weaker constraint. The definition of *select* is a parameter of the engine, and we say more about it later.

We are now in a position to define the main step of our engine. A cluster is said to be in *substitution form* if it contains only approximate, terminal and residual groups. Then:

Definition 2 (Unfolding Step) *Let $\mathcal{C}_1$ be a cluster containing a non-terminal group $\mathcal{G}$. Let $\mathcal{C}_2$ be a cluster in substitution form. Construct a new cluster by replacing $\mathcal{G}$ in $\mathcal{C}_1$ by all of the following groups:*

- $\mathcal{G}^{®}$ *, that is, the group $\mathcal{G}$ marked as* <u>*residual*</u>*;*
- $\mathcal{G} \circ [\mathcal{A}]$*, for all approximate groups $[\mathcal{A}] \in \mathcal{C}_2$, and*
- $\mathcal{G} \diamond [c]_i$*, for all exact (i.e. terminal or residual) groups $[c]_i \in \mathcal{C}_2$.*

We say that this new cluster is the result of <u>substituting</u> $\mathcal{C}_2$ <u>into</u> $\mathcal{C}_1$ <u>at</u> $\mathcal{G}$. □

Example 1 Consider the abstract domain *LSign*. The following presents clusters $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ respectively. $\mathcal{C}_3$ is the result of substituting $\mathcal{C}_2$ into $\mathcal{C}_1$.

$$\left(\begin{bmatrix} X^{\uparrow} = U^{\downarrow} + 1 \\ Y^{\uparrow} = V^{\downarrow} - 2 \end{bmatrix}_i \right) \quad \left(\begin{bmatrix} 5 \times U^{\uparrow} + 9 \times V^{\uparrow} = 20 \\ U^{\uparrow} \geq 0 \end{bmatrix}_{\star} \right) \quad \left(\begin{array}{c} \begin{bmatrix} X^{\uparrow} = U^{\downarrow} + 1 \\ Y^{\uparrow} = V^{\downarrow} - 2 \end{bmatrix}_i^{®} \\ \begin{bmatrix} X^{\uparrow} \geq 1 \end{bmatrix}_{\star} \\ \begin{bmatrix} \oplus X \oplus Y = \oplus \end{bmatrix} \end{array} \right)$$

Consider $\mathcal{C}_3$. The group in $\mathcal{C}_1$ becomes residual in $\mathcal{C}_3$. The group $[X^{\uparrow} \geq 1]_{\star}$ in $\mathcal{C}_3$ is obtained from *select*, as a truncation of the composition

$$\begin{bmatrix} X^{\uparrow} = U^{\downarrow} + 1 \\ Y^{\uparrow} = V^{\downarrow} - 2 \end{bmatrix}_i \circ \begin{bmatrix} 5 \times U^{\uparrow} + 9 \times V^{\uparrow} = 20 \\ U^{\uparrow} \geq 0 \end{bmatrix}_{\star} = \begin{bmatrix} 5 \times X^{\uparrow} + 9 \times Y^{\uparrow} = 7 \\ X^{\uparrow} \geq 1 \end{bmatrix}_{\star}$$

Note that there are really many ways to write the constraints resulting from this composition. The behavior of *select* will depend on which "canonical form" is chosen. We discuss this issue in the next section. Now the third group $[\oplus X \oplus Y = \oplus]$ is approximate, and is obtained by applying ABS to this composition above. □

An unfolding step is applied to a collection of semantic equations as follows. Let $\mathcal{C}_1$ appear in the semantic equation for Ψ_i, and let $[c]_j \in \mathcal{C}_1$. Let $\mathcal{C}_2$ appear in the semantic equation for Ψ_j. If substituting $\mathcal{C}_2$ into $\mathcal{C}_1$ at $[c]_j$ results in a cluster $\mathcal{C}$ that does not already appear in the semantic equation for Ψ_i, then $\mathcal{C}$ is added to this equation and we say that the unfolding step was *effective*.

[3] Note that unlike examples in the previous section, we do not require j to be $\star$ in this definition. This extension relates to the use of residual groups, described later.

input a program P;
obtain the semantic equations associated with P;
repeat
    apply an effective unfolding step if possible;
until no effective unfolding step is possible;
delete all clusters not in substitution form;
delete all residual groups[4];
output the resulting equations;

Figure 1: **The Unfolding Engine**

The complete engine is presented in Figure 1. It is essentially an exhaustive application of the unfolding step just defined. When no more effective unfolding steps can be performed, all clusters in substitution form are output after first deleting the residual groups therein. Such clusters represent fixed sets of constraints because the groups therein are not recursively defined in terms of other groups and/or Ψ_i variables. The important point here is that the output is an *explicit* representation of a set of constraints for each Ψ_i. A post-processing phase may then be applied to extract the specific information sought.

Example 2 To exemplify the use of residual groups, consider the following CLP($\mathcal{R}$) program, and its semantic equations (after some basic simplification):

```
p(X, Y) :- q(f(X, Y)), s(Y).
q(U) :- r(U).
r(f(V, W)) :- V + W = 3.
s(Z) :- Z = 1.
```

$$\Psi_1 = \begin{pmatrix} [f(X^{\uparrow}, Y^{\uparrow}) = U^{\downarrow}]_2 \\ [Y^{\uparrow} = Z^{\downarrow}]_4 \end{pmatrix}^{\textcircled{1}}$$
$$\Psi_2 = ([U^{\uparrow} = f(V^{\downarrow}, W^{\downarrow})]_3)^{\textcircled{2}}$$
$$\Psi_3 = ([V^{\uparrow} + W^{\uparrow} = 3]_{\star})^{\textcircled{3}}$$
$$\Psi_4 = ([Z^{\uparrow} = 1]_{\star})^{\textcircled{4}}$$

Now consider the execution of the engine, again using the *LSign* domain. The initial semantic equations contain two clusters in substitution form: ③ and ④. The following two unfolding steps can be performed: cluster ③ can be substituted into ② and cluster ④ can be substituted into ①, yielding

$$\begin{pmatrix} [U^{\uparrow} = f(V^{\downarrow}, W^{\downarrow})]_3^{\textcircled{R}} \\ [\top]_{\star} \end{pmatrix}^{\textcircled{5}} \qquad \begin{pmatrix} [f(X^{\uparrow}, Y^{\uparrow}) = U^{\downarrow}]_2 \\ [Y^{\uparrow} = 1]_{\star} \end{pmatrix}^{\textcircled{6}}$$

where $\top$ is the least informative abstract formula. Thus ⑤, as a result of approximation and *select*, provides no new information, except that it now contains a residual group. Note also that we have omitted residual and approximate groups from ⑥ because they would have been redundant; *select*, in this case, did not change its input constraints. Now, these new clusters are added to the equations for Ψ_2 and Ψ_1 respectively. Next, cluster ⑤ is substituted into ⑥ at the first group, and this yields

[4] We remark that these groups in fact do contain additional information. More complex post-processing algorithms can be designed to take advantage of this.

$$\begin{pmatrix} [f(X^{\uparrow}, Y^{\uparrow}) = U^{\downarrow}]_2^{\textcircled{R}} \\ [X^{\uparrow} = V^{\downarrow} \ \wedge \ Y^{\uparrow} = W^{\downarrow}]_3 \\ [\top]_\star \\ [Y^{\uparrow} = 1]_\star \end{pmatrix}^{⑦}$$

which is added to the equation for Ψ_1. Finally, cluster ③ can be substituted into ⑦ at the second group to produce ⑧, which in turn can be simplified into ⑨.

$$\begin{pmatrix} [f(X^{\uparrow}, Y^{\uparrow}) = U^{\downarrow}]_2^{\textcircled{R}} \\ [X^{\uparrow} + Y^{\uparrow} = 3]_\star \\ [\top]_\star \\ [Y^{\uparrow} = 1]_\star \end{pmatrix}^{⑧} \qquad \begin{pmatrix} [f(X^{\uparrow}, Y^{\uparrow}) = U^{\downarrow}]_2^{\textcircled{R}} \\ [X^{\uparrow} = 2 \ \wedge \ Y^{\uparrow} = 1]_\star \end{pmatrix}^{⑨}$$

Note that the residual group was used maintain the connection between X, Y and U, because the abstract domain was not sufficiently expressive to represent the constraints $U^{\uparrow} = f(V^{\downarrow}, W^{\downarrow}) \ \wedge \ X^{\uparrow} + Y^{\uparrow} = 3$.

At this point the engine terminates, and outputs all clusters in substitution form: ③, ④, ⑤ and ⑧. The output of interest is ⑧, corresponding to Ψ_1, which represents the constraint $X = 2 \ \wedge \ Y = 1$. $\square$

For this example we could make *ad hoc* extensions to the domain to increase its expressiveness: for example, consider a domain that combines a domain for representing term structure with *LSign* so that we can represent linear constraints between sub-expressions of a structure. However, not only is such a domain computationally expensive, but it also limited: we can always construct a program that exceeds the inherent bounds of the domain. In contrast, residuals essentially borrow from set constraints the ability to represent unbounded properties of program structure.

6 Termination: Solved Forms and Select

To simplify our presentation so far, we have treated semantically equivalent constraints as equal. Moreover, we have freely allowed constraints to be rewritten into an equivalent but simpler form (for example, constraints such as $f(X, Y) = f(U, V)$ have been rewritten into $X = U \ \wedge \ Y = V$). However, such an abstract view of the unfolding engine glosses over an important issue: how are constraints represented and manipulated by the algorithm? This question has particular relevance to the *select* function, whose definition is typically in terms of some notion of growth of constraints during the substitution step. Such notions cannot be formalized without fixing the constraint representation.

We address this issue by explicitly introducing a constraints simplification procedure *simplify*, which is invoked on each constraint considered by the algorithm. Specifically, we shall invoke *simplify* on every constraint in the initial semantic equations, and on every new constraint constructed during the execution of the algorithm. For example, when we construct new constraints by computing $[c_1]_i \circ [c_2]_j$, we shall actually compute

$$\left[\mathit{simplify} \left((\exists_S)(c_1{\uparrow} \wedge c_2{\downarrow}) \right) \right]_j$$

rather than $[(\exists_S)(c_1{\uparrow} \wedge c_2{\downarrow})]_j$. In effect, *simplify* defines a notion of "solved form" constraints, and these solved form constraints are the basis of our constraint representation: the algorithm maintains all constraints in solved form.

Typically, we will choose *simplify* to be a suitable constraint solver for the constraint domain at hand. For the domain of finite trees, it would be appropriate to use a unification algorithm (in effect this is what is used in [6]). For the domain of infinite trees, one would use an infinite tree unification algorithm. For linear equalities, a version of Gaussian Elimination could be appropriate. For more complex constraint domains, such as that of CLP($\mathcal{R}$), the choices are less obvious. We could for example use a complete solver for finite trees and real numbers, but this would be expensive. At the other end of the spectrum, we could use a trivial solver: the identity function. This would be cheap, but new constraints would always grow and so the accuracy of the algorithm would more likely suffer (because of *select*). There is an obvious accuracy/efficiency tradeoff here.

However, more important than this tradeoff is the intended purpose of the analysis. If the analysis is used to obtain general semantic properties about a program, then any *simplify* is appropriate. On the other hand, if we are using the analysis for the purposes of optimizing the execution of a program on a particular CLP system, then it is important to choose *simplify* to be compatible with the CLP system at hand. For example, in the context of the CLP($\mathcal{R}$) system [8], it would be appropriate for *simplify* to solve tree equations, linear equations and linear inequalities, but not arbitrary non-linear constraints. In general, we want *simplify* to be weaker than the solver of the CLP system.

We next discuss the *select* operation. It is a parameter to our engine, and it serves to ensure termination. Ideally, *select* is the identity function; in reality, this choice may not be possible nor practical. The appropriate choice is clearly dependent on many factors, such as the underlying constraint and abstract domains, the runtime properties sought, and of course requirements on accuracy and efficiency. For this paper, we simply require that

- *select* returns a weaker constraint, that is, $c \models select_{c_1,c_2}(c)$ for all c, c_1, c_2;
- the set of constraints returned during execution of the engine is bounded.

For example, an intuitive way of defining *select* is via a metric ∇ on constraints that satisfies the following condition: given a constraint c, there are only a finite number of constraints c' such that $var(c') \subseteq var(c)$ and $\nabla(c') \leq \nabla(c)$. For equations over term structures, ∇ can just count the symbols appearing in the constraints (c.f. [6]). However, for constraints such as those in CLP($\mathcal{R}$), there can be infinite collections of constraints of the form $\{X = 1, X = 2, X = 3, \ldots\}$ and so here ∇ must also take into account the *size* of constants appearing in constraints. Now, *select* can be any function such that

$$\nabla(select_{c_1,c_2}(c)) \leq max(\nabla(c_1), \nabla(c_2)).$$

For example, $select_{c_1,c_2}(c)$ can simply truncate c so that ∇ is sufficiently reduced.

7 Main Results

The first result address the correctness of the algorithm. The proof, omitted for space reasons, has two parts. The first shows that the algorithm terminates. The second shows that, on termination, the unfolding transformation has been applied sufficiently often so that the output clusters contain a conservative (superset) approximation of the least model of the original semantic equations. The second result addresses the relative accuracy of the algorithm.

Theorem 7.1 (Correctness and Termination) *Let P be a program and let $\mathcal{D}$ be a finite*[5] *abstract domain. Then the unfolding engine terminates on P using $\mathcal{D}$, and the output semantic equations are a conservative approximation of the initial semantic equations for P.* □

Next we compare our algorithm with those in the literature. Now, it is clearly difficult to characterize every algorithm for logic program analysis using a single algorithm. This is because each algorithm is designed with different criteria — *ad hoc* approximations are often introduced for efficiency reasons. However, these differences are rarely fundamental from a conceptual point of view. Consequently, it is possible to define an idealized engine which encompasses the essence of the abstract interpretation technique that underlies these algorithms, and is at least as accurate as any of them. We call this idealized engine the *standard* engine, and it can be argued that this algorithm is the most obvious embodiment of an abstract domain into an analyzer. More detailed formalizations of this engine can be obtained from [10] and [2].

In general, each analysis algorithm starts by associating a value from the chosen abstract domain $\mathcal{D}$ with designated parts of the program. The operation of the algorithm then consists of repeatedly recomputing each value from the values previously computed. In what follows, we give, for the sake of completeness, an informal definition of the standard engine for *bottom-up* analysis; other analyses, such as top-down, can be similarly described.

```
Initialize the array abs_formula[1..N];
loop {
      for (1 ≤ i ≤ N) {
            let c denote the constraints of rule i;
            let B̃ denote the body atoms of rule i;
            new_abs_formula[i] := ⋃_H̃ D-solve(c, B̃, H̃, abs_formula[])
                  where H̃ ranges over sequences of rule heads
                  matching B̃;
      }
      if (new_abs_formula[] = abs_formula[]) break;
      abs_formula[] := new_abs_formula[];
}
output abs_formula[];
```

where the operation $\mathcal{D}$-*solve* uses the abstraction and concretization functions associated with $\mathcal{D}$ to implement the basic operational step of obtaining a new abstract formula from a given old abstract formula and a new (concrete) constraint. Formally, let $\tilde{B}$ be $B_1, \cdots, B_m$, and let $\tilde{H}$ be $H_{j_1}, \cdots, H_{j_m}$.

Then $\mathcal{D}\text{-}solve(c, \tilde{B}, \tilde{H}, abs\_formula[])$ is defined to be

$$\text{ABS}\left(\left\{ c \wedge \left(\begin{array}{c} \exists_{\overline{var(B_1)}}(B_1 = H_{j_1}\theta \wedge c_1\theta) \\ \wedge \cdots \wedge \\ \exists_{\overline{var(B_m)}}(B_m = H_{j_m}\theta \wedge c_m\theta) \end{array} \right) : \begin{array}{l} c_i \in \text{CONC}(abs\_formula[j_i]), \\ 1 \leq i \leq m \end{array} \right\}\right)$$

where θ renames away from $B_1, \ldots, B_m$.

[5] Some algorithms use domains satisfying a weaker condition such as the "finite chain" property. Our engine can in general be adapted to terminate on such domains. We omit the details for simplicity.

In order to compare the standard engine with the unfolding engine, we now formulate the operation of the standard engine using our framework of groups clusters, composition etc. Recall that in our framework, we can perform bottom-up or top-down analysis simply by using the appropriate semantic equations. Now, the standard engine proceeds exactly as the unfolding engine, except that:

- all terminal groups $[c]$ are initially replaced by $\text{ABS}(c)$;
- the unfolding step is simplified so that no residual groups are produced.

The main effect of these changes is that clusters in substitution form contain only approximate groups (and no terminal or residual groups). Hence the only operation required during unfolding is the composition $[\mathcal{E}] \circ [\mathcal{A}]$ of an exact and approximate group.

Proposition 2 *This formulation correctly describes the standard engine.* □

Clearly these modified unfolding steps are less accurate than the unfolding step used in the (full) unfolding engine. Hence we can verify that

Theorem 7.2 *The unfolding engine is uniformly more accurate than the standard engine in the sense: given a finite abstract domain, the output of the unfolding engine, for any program, is more accurate than that of the standard engine.* □

8 Conclusion

We presented a generic engine for CLP analysis, based on unfolding. To highlight its advantages, we now make two comparison. The first is to an earlier unfolding engine for traditional logic programs [6]. The new engine represents a substantial advance because:

- The old engine was designed around a specific representation of substitutions that cannot be generalized to CLP. In contrast, the new engine is based on a very general and uniform treatment of constraints.
- The new engine is more efficient, conceptually simpler and avoids some *ad hoc* steps contained in the previous engine.
- When used to analyze logic programs, the new engine is strictly more accurate than the old engine. This is because residuals are now represented as constraints and hence can be "bi-directional" in the sense that they can transfer structural information in both directions. In contrast, the previous engine essentially truncated residual groups so that they were essentially substitutions mapping input variables into output variables. For example, the old engine would truncate the residual group $[X_1{\uparrow} = f(Y_1{\downarrow}), f(X_2{\uparrow}) = Y_2{\downarrow}]^{R}$ into $[X_1{\uparrow} = f(Y_1{\downarrow})]^{R}$, whereas the new engine maintains the entire residual group (since it does not rely on using substitution forms).

Our second comparison is with traditional CLP analysis. The framework of the new engine is conceptually simple and yet general; it includes the standard CLP engine as a special case. The new engine is also flexible, and provides new new opportunities for tuning efficiency/accuracy tradeoffs. By varying aspects of the engine, one can obtain a wide spectrum of behaviors. Finally, regardless of tuning, constraint domain, or abstract domain, the new unfolding engine is uniformly more accurate than the traditional CLP engine.

References

[1] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", *Proc.* 4^{th} *ACM-POPL*, 1977, pp 238–252.

[2] M. Garcia de la Banda and M. Hermenegildo, "A Practical Approach to the Global Analysis of CLP Programs", *Proc. ILPS*, 1993, pp 437–455.

[3] N. Heintze, "Set Based Program Analysis", Ph.D. thesis, School of Computer Science, Carnegie Mellon University, October 1992.

[4] N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs", *Proc.* 17^{th} *ACM-POPL*, 1990, pp 197–209.

[5] N. Heintze and J. Jaffar, "A Decision Procedure for a Class of Herbrand Set Constraints", *Proceedings* 5^{th} *IEEE-LICS*, 1990, pp 42–51.

[6] N. Heintze and J. Jaffar, "An Engine for Logic Program Analysis", *Proc.* 7^{th} *IEEE-LICS*, pp 318–328, 1992.

[7] J. Jaffar and J.-L. Lassez, "Constraint Logic Programming", *Proc.* 14^{th} *ACM-POPL*, 1990, pp 111–119.

[8] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap, "The CLP($\mathcal{R}$) Language and System", *ACM TOPLAS*, 14(3), pp 339–395, 1992.

[9] N.D. Jones and H. Søndergaard, "A Semantics-based Framework for the Abstract Interpretation of PROLOG", in *Abstract Interpretation of Declarative Languages*, (S. Abramsky and C. Hankin, Eds), Ellis Horwood, 1987, pp 123–142.

[10] K. Marriott and H. Søndergaard, "Analysis of Constraint Logic Programs", *Proc. 1990 North American Conf. on Logic Programming*, 531-547, 1990.

[11] K. Marriot and P. Stuckey, "The 3 R's of Optimizing Constraint Logic Programs: Refinment, Removal and Reordering", *Proc.* 20^{th} *ACM-POPL*, 1993, pp 334–344.

[12] K. Marriot and P. Stuckey, "LSign: Approximating Possible Interaction between Linear Arithmetic Constraints", *Proc. ILPS*, 1994.