Analysis of normal logic programs

François Fages and Roberta Gori

LIENS CNRS, Ecole Normale Supérieure,
45 rue d'Ulm, 75005 Paris, France,
E-mail: `fages@dmi.ens.fr`

Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa, Italy
E-mail: `gori@di.unipi.it`
Ph.: +39-50-887248 Fax: +39-50-887226

Abstract. In this paper we present a dataflow analysis method for normal logic programs interpreted with negation as failure or constructive negation. We apply our method to a well known analysis for logic programs: the depth(k) analysis for approximating the set of computed answers. The analysis is correct w.r.t. SLDNF resolution and optimal w.r.t. constructive negation.

Keywords: *Abstract interpretation, static analysis, logic programming, constructive negation.*

1 Introduction

Important results have been achieved for static analysis using the theory of abstract interpretation [6]. Abstract interpretation is a general theory for specifying and validating program analysis.

A key point in abstract interpretation is the choice of a reference semantics from which one can abstract the properties of interest. While it is always possible to use the operational semantics, it is possible to get rid of useless details, by choosing a more abstract semantics as reference semantics. In the case of definite logic programs, much work has been done in this sense. Choosing the most abstract logical least model semantics limits the analysis to type inference properties, that approximate the ground success set. Non-ground model semantics have thus been developed, under the name of S-semantics [2], and proved useful for a wide variety of goal-independent analysis ranging from groundness, to sharing, call patterns, etc. All the intermediate fixpoint semantics between the most abstract logical one and the most concrete operational one, form in fact a hierarchy related by abstract interpretation, in which one can define a notion of the best reference semantics [12] for a given analysis.

On the other hand, less work has been done on the analysis of normal logic programs, although the finite failure principle, and hence SLDNF resolution, are standard practice. The most significant paper on the analysis of normal logic programs, using the theory of abstract interpretation, is the one by Marriott and Søndergaard [19], which proposes a framework based on Fitting's least three-valued model semantics [11]. Since this reference semantics is a ground semantics, the main application of this framework is type analysis. Marriott and Søndergaard already pointed out that a choice of a different reference semantics could lead to an improved analysis. Fitting's least three-valued model semantics is, in fact, an abstraction (a non recursively enumerable one, yet easier to define) of Kunen's three-valued logical semantics [14] which is more faithful to SLDNF resolution [15] and complete w.r.t. constructive negation.

These are exactly the directions along which we try to improve the results of Marriott and Søndergaard. We consider the inference rule of constructive negation, which provides normal logic programs with a sound and complete [21] operational semantics w.r.t. Kunen's logical semantics [14]. We propose an analysis method for normal logic programs interpreted with constructive negation, based on the generalized S-semantics given in [9] and on the hierarchy described in [10]. We present here an analysis based on the depth(k) domain which approximates the computed answers obtained by constructive negation and therefore the three-valued consequences of the program completion and CET (Clark's equational theory). Other well known analyses for logic programs can be extended to normal logic programs. For example, starting from a suitable version of Clark's semantics a groundness analysis was defined which is correct and optimal w.r.t. constructive negation. Here, for lack of space, we present only the depth(k) analysis. We show that it is correct and also optimal w.r.t. constructive negation. Finally we give an example which shows that in the case of type inference properties our semantics yields a result which is more precise than the one obtained by Marriott and Søndergaard.

¿From the technical point of view, the contribution of the paper is the definition of a normal form for first order constraints on the Herbrand Universe, which is suitable for analysis. In fact the normal form allows us to define an abstraction function which is a congruence w.r.t. the equivalence on constraints induced by the Clark's equality theory.

The paper is organized as follows. In section 2 we introduce some preliminary notions on constructive negation. In section 3 we define a *normal* form on the concrete domain of constraints in order to deal, with equivalence classes of constraints w.r.t. the Clark's equational theory. Section 4 defines the abstract domain and abstract operator and show its correctness and optimality (under suitable assumptions on the depth of the cut) w.r.t. the concrete one. Finally, subsection 4.5 shows an example.

2 Preliminaries

The reader is assumed to be familiar with the terminology of and the basic results in the semantics of logic programs [1,17] and with the theory of abstract interpretation as presented in [6,7].

2.1 Normal logic programs and constructive negation

We consider the equational version of normal logic programs, where a *normal program* is a finite set of clauses of the form $A \leftarrow c|L_1, ..., L_n$, where $n \geq 0$, A is an atom, called the head, c is a conjunction of equalities, and $L_1, ..., L_n$ are literals. The *local variables* of a program clause are the free variables in the clause which do not occur in the head. With $Var(A)$ we intend the free variables in the atom A.

In order to deal with constructive negation, we need to consider the domain $\mathcal{C}$ of full first-order equality constraints on the structure $\mathcal{H}$ of the Herbrand domain. Assuming an infinite number of function symbols in the alphabet, *Clark's equational theory* (CET) provides a complete decidable theory for the constraint language [18,14], i.e.

1. (soundness) $\mathcal{H} \models CET$,
2. (completeness) for any constraint c, either $CET \models \exists c$ or $CET \models \neg\exists c$.

A constraint is in *prenex form* if all its quantifiers are in the head. The set of free variables in a constraint c is denoted by $Var(c)$. For a constraint c, we shall use the notation $\exists c$ (resp. $\forall c$) to represent the constraint $\exists X\ c$ (resp. $\forall X\ c$) where $X = Var(c)$.

A *constrained atom* is a pair $c|A$ where c is an $\mathcal{H}$-solvable constraint such that $Var(c) \subseteq Var(A)$. The set of constrained atoms is denoted by B. A *constrained interpretation* is a subset of B. A *three-valued* or *partial constrained interpretation* is a pair of constrained interpretations $< I^+, I^- >$, representing the constrained atoms which are true and false respectively (note that because of our interest in abstract interpretations we do not impose any consistency condition between I^+ and I^-).

Constructive negation is a rule of inference introduced by Chan for normal logic programs in [3], which provides normal logic programs with a sound and complete [21] operational semantics w.r.t. Kunen's logical semantics [14]. In logic programming, Kunen's semantics is simply the set of three-valued consequences of the program's completion and the theory CET.
The S-semantics of definite logic programs [2] has been generalized to normal logic programs in [9] for a version of constructive negation, called constructive negation by pruning. The idea of the fixpoint operator, which captures the set of computed answer constraints, is to consider a *non-ground finitary* (hence continuous) version of Fitting's operator. Here we give a definition of the operator $T_P^{\mathcal{B}_\mathcal{D}}$ which is parametric w.r.t. the domain $\mathcal{B}_\mathcal{D}$ of constrained atoms and the operations on constraints on the domain $\mathcal{D}$.

Definition 1. Let P be a normal logic program. $T_P^{\mathcal{B}_\mathcal{D}}$ is an operator over $\mathcal{P}(\mathcal{B}_\mathcal{D}) \times \mathcal{P}(\mathcal{B}_\mathcal{D})$ defined by

$$\begin{array}{ll}
T_P^{\mathcal{B}_\mathcal{D}}(I)^+ = \{c|p(X) \in \mathcal{B}_\mathcal{D} : & \text{there exists a clause in } P \text{ with local variables } Y, \\
& C = p(X) \leftarrow d|A_1, ..., A_m, \neg A_{m+1}, ..., \neg A_n. \\
& c_1|A_1, ..., c_m|A_m \in I^+,\ c_{m+1}|A_{m+1}, ..., c_n|A_n \in I^- \\
& \text{such that } c = \overline{\overline{\exists}} Y\ (d\ \overline{\wedge} c_1 \overline{\wedge} \ldots \overline{\wedge} c_n)\} \\
T_P^{\mathcal{B}_\mathcal{D}}(I)^- = \{c|p(X) \in \mathcal{B}_\mathcal{D} : & \text{for each clause defining } p \text{ in } P \text{ with loc. var. } Y_k, \\
& C_k = p(X) \leftarrow d_k|A_{k,1}, ..., A_{k,m_k}, \beta_k. \\
& \text{there exist } e_{k,1}|A_{k,1}, ..., e_{k,m_k}|A_{k,m_k} \in I^-, \\
& n_k \geq m_k,\ e_{k,m_{k+1}}|A_{k,m_{k+1}}, ..., e_{k,n_k}|A_{k,n_k} \in I^+, \\
& \text{where for } m_{k+1} \leq j \leq n_k,\ \neg A_{k,j} \text{ occurs in } \beta_k, \\
& \text{such that } c = \overline{\bigwedge}_k\ \overline{\forall} Y_k\ (\overline{\neg}\ d_k \overline{\vee}\ e_{k,1} \ldots \overline{\vee}\ e_{k,n_k})\}.
\end{array}$$

where the operations $\overline{\overline{\exists}}, \overline{\forall}, \overline{\neg}, \overline{\vee}, \overline{\wedge}$, are the corresponding operations on the constraint domain of $\mathcal{D}$.

In the case of a normal logic program, the operator T_P^B defines a generalized S-semantics which is fully abstract w.r.t. the computed answer constraints with constructive negation by pruning [9]. By soundness it approximates also the set of computed answer constraints under the SLDNF resolution rule, or under the Prolog strategy.

In [10] we have shown that this operator defines a hierarchy of reference semantics related by abstract interpretation, that extends the hierarchy defined by Giacobazzi for definite logic programs [12]. Here we show the use of the hierarchy for the static analysis of normal logic programs.

3 Normal forms in CET

Unlike the semantics in Marriott and Søndergaard's framework, our reference semantics is a non ground semantics and has to deal with first-order equality constraints. The first problem that arises is to define a normal form for such constraints on the Herbrand domain, so that abstraction functions on constrained atoms can be defined. In general, in fact, given a theory th, we are interested in working with equivalence classes of constraints w.r.t. the equivalence of the constraints in th. Namely c is equivalent to c' if $th \models c \leftrightarrow c'$. Therefore we need the abstraction function on the concrete constraint domain to be a congruence. This is a necessary property since it permits to be independent from the syntactic form of the constraints.

Dealing with normal logic programs, we need to achieve this property in CET. We thus need to introduce a normal form for first-order equality constraints, in a similar way to what has been done for the analysis of definite programs where the normal form is the unification solved form [16]. Here we shall define a new notion of "false-simplified" normal forms, making use of Colmerauer's solved forms for inequalities [4], Maher's transformations for first-order constraints [18] and an extended disjunctive normal form [13].

First let us motivate the need of a "false-simplified" form. Let us call a *basic constraint* an equality or an inequality between a variable and a term. The abstraction function will be defined inductively on the basic constraints, and it will sometimes (e.g. for groundness analysis) abstract to *true* some inequalities. Consider, for example, the following constraint $d = \forall X(Y = b \wedge X \neq f(a))$. d is $\mathcal{H}$-equivalent to $false$. If the abstraction of $X \neq f(a)$ is *true* then the abstraction of d will be the abstraction of $Y = b$, which cannot be $\mathcal{H}$-equivalent to the abstraction of $false$. Therefore we need to define a normal form where the constraints which are $\mathcal{H}$-equivalent to $false$, are eliminated.

Definition 2. Consider a constraint d in prenex disjunctive form, $d = \Delta(\vee_i A_i)$, where Δ is a sequence of quantified variables and $\vee_i A_i$ is a finite disjunction. d is in a $false$-simplified form if, either there does not exist a proper subset I of the $i's$ such that $\mathcal{H} \models \Delta(\vee_i A_i) \leftrightarrow \Delta(\vee_{i \in I} A_i)$, or such an I exists and there exists also a subset K of I, such that $\vee_{j \notin I} A_j$ is $\mathcal{H}$-equivalent to $\vee_{k \in K} A_k$.

The latter condition assures that we really eliminate constraints that are $\mathcal{H}$-equivalent to $false$ and that are not just redundant in the constraint. Now the existence of a *false-simplified* form for any constraint can be proved simply with the following:

Algorithm 3. Input: a constraint in prenex disjunctive form $d = \Delta(\vee_i A_i)$. Let us call U the set of the indices i's in $d = \Delta(\vee_i A_i)$.

1. Let I and J be the partition of U such that $i \in I$ if $\mathcal{H} \models \exists\Delta(A_i)$, otherwise $i \in J$.
2. Repeat $I := I \cup S$ as long as there exists an $S \subseteq J$ such that $\mathcal{H} \models \exists\Delta(\vee_{i \in S} A_i)$ and for all $j \in S$ $\mathcal{H} \not\models \exists\Delta(\vee_{i \in (S \setminus \{j\})} A_i)$.
3. Let $S \subseteq J \setminus I$ be any minimal set such that $\mathcal{H} \models \exists\Delta(\vee_{s \in S} A_s \vee_{i \in I} A_i)$ and $\mathcal{H} \models \Delta(\vee_{s \in S} A_s \vee_{i \in I} A_i) \leftrightarrow d$, do $I := I \cup S$,
4. Output: $\Delta(\vee_{i \in I} A_i)$.

The idea of the algorithm is to find a subset of the conjunctions A_i's (those with $i \in I$) such that $\Delta(\vee_{i \in I} A_i)$ is in *false-simplified* form and it is $\mathcal{H}$-equivalent to $\Delta(\vee_i A_i)$. In the first step we select the A_i's such that $\Delta(A_i)$ is $\mathcal{H}$-satisfiable. In this case, in fact, A_i cannot be $\mathcal{H}$-equivalent to $false$ and it can be put in the set I. In the second step from the remaining A_i's we select the set of A_i's such that their Δ quantified disjunction is $\mathcal{H}$-satisfiable, since we check that all the A_i's are necessary for the quantified disjunction to be $\mathcal{H}$-satisfiable, the considered A_i's can not be $\mathcal{H}$-equivalent to $false$. At the end of this process, if the resulting constraint is $\mathcal{H}$-equivalent to the input constraint, we stop. Otherwise, we add a minimum number of the not yet selected A_i's such the $\Delta(\vee_i A_i)$ for the selected i's is $\mathcal{H}$-equivalent to the input constraint. Since we add a minimum number of not yet selected A_i's, we are sure that the resulting constraint is in *false-simplified* form. Example 4 shows how the algorithm 3 works on two examples.

Example 4. 1. Input: $c_1 = \forall T(A_1 \vee A_2 \vee A_3 \vee A_4)$, $A_1 = (T = f(H) \wedge Y = a)$, $A_2 = (T \neq f(a) \wedge Y = b)$, $A_3 = (Y \neq g(H,T))$, $A_4 = (T \neq a \wedge Y = a)$.
$I_1 = \{3\}$.
$I_2 = \{3, 1, 2\}$.
$I_3 = \{3, 1, 2\}$(since $\mathcal{H} \models \forall T(\vee_{i \in I_2} A_i) \leftrightarrow c_1$).
Output: $\forall T(A_1 \vee A_2 \vee A_3)$.

2. Input: $c_2 = \forall T(A'_1 \vee A'_2 \vee A'_3)$, $A'_1 = (T \neq f(U) \wedge T \neq f(V))$,
   $A'_2 = (T = H)$, $A'_3 = (U \neq V \wedge T = f(a))$.
   $I_1 = \{\}$.
   $I_2 = \{1,2\}$.
   $I_3 = \{1,2,3\}$(since $\mathcal{H} \not\models \forall T(\vee_{i \in I_2} A'_i) \leftrightarrow c_2$).[1]
   Output: $\forall T(A'_1 \vee A'_2 \vee A'_3)$.

Theorem 5. *For any input constraint $c = \Delta(\vee_i A_i)$, algorithm 3 terminates and computes a* false-simplified *form logically equivalent to c.*

Note that all the false-simplified forms of a constraint c are $\mathcal{H}$-equivalent.

Now the intuitive idea for a normal form is the following. We put a constraint in prenex form and we compute the disjunctive form of its quantifier free part. We make equality and inequality constraints interact in every conjunction of the disjunctive form and then we compute the false-simplified form for the resulting constraint. The problem is that if we consider a standard disjunctive normal form, we would not be able to see explicitly all the relations between constraints in disjunctions. Consider, for example, the constraint $(X = f(H) \vee (H \neq f(a))$. This constraint is equivalent, therefore $\mathcal{H}$-equivalent, to the constraint $((X = f(f(a)) \wedge H = f(a)) \vee H \neq f(a))$. Note that the equality $H = f(a)$ is not explicit in the first disjunction. Since the abstraction function will act on the terms of the disjunction independently, this could cause a problem. This is why we will use a well known *extended disjunctive form* defined for Boolean algebra and applied, in our case, to the algebra of quantifier free constraints.
In the next theorem with B_i we denote basic equality or inequality constraints $(X = t$ or $X \neq t)$. For any B_i let ${B_i}^{false} = \neg B_i$ and ${B_i}^{true} = B_i$.

Theorem 6. *[13] For every Boolean formula ϕ on basic equality or inequality constraints $B_1, \ldots, B_n$,*
$\phi \leftrightarrow \psi$ where $\psi = (\bigvee_{(a_1,\ldots,a_n) \in \{false,true\}^n} \phi(a_1,\ldots,a_n) \wedge B_1^{a_1} \wedge \ldots \wedge B_n^{a_n})$.

Note that ψ is a formula in disjunctive form. ψ has in fact a particular disjunctive form where each conjunction contains all the basic constraints (possibly negated) of ϕ. This is why, this form is able to capture all the possible relations between the different terms of a disjunction.

We will call the formula ψ the *extended disjunctive normal form (dnf)* of ϕ. The next example shows how the extended disjunctive normal form works on a constraint c_1.

Example 7. $c_1 = (X = f(H) \vee H \neq f(a))$.
$dnf(c_1) = \ ((X = f(f(a)) \wedge H = f(a)) \vee$
$\qquad (X = f(H) \wedge H \neq f(a)) \vee (X \neq f(H) \wedge H \neq f(a)))$.

Note that although c_1, $dnf(c_1)$ and $((X = f(f(a)) \wedge H = f(a)) \vee H \neq f(a))$ are $\mathcal{H}$-equivalent, $dnf(c_1)$ is the most "complete" in the sense that it shows syntactically all the relations between constraints in disjunctions.

[1] $U = b, H = f(b), V = a$, in fact, is an assignment (for the free variables of c_2), which is a solution of c_2 but is not a solution of $\forall T(\vee_{i \in I_2} A'_i)$.

We can now define the normal form, $Res(c)$, of a first-order equality constraint c, as the result of the following steps:

1. Put the constraint c in prenex form obtaining $\Delta(c_1)$, where Δ is a sequence of quantified variables and c_1 is the quantifier free part of c.
2. Compute $dnf(c_1) = \vee(A_i)$,
3. Simplify each conjunction A_i obtaining $A'_i = ResConj(A_i)$,
4. Return a $false$-simplified form of the constraint $\Delta(\vee A'_i)$.

where the procedure for simplifying each conjunction is based on Maher's canonical form [18] and Colmerauer's simplification algorithm for inequalities [4]. The procedure performs the following steps,

$$ResConj(A)$$

1. compute a unification solved form for the equalities in the conjunction A
2. for each equality $X = t$ in A, substitute X by t at each occurrence of X in the inequalities of conjunction A.
3. simplify the inequalities by applying the following rules,
   (a) replace $f(t_1, \ldots, t_n) \neq f(s_1, \ldots, s_n)$ by $t_1 \neq s_1 \vee \ldots \vee t_n \neq s_n$.
   (b) replace $f(t_1, \ldots, t_n) \neq g(s_1, \ldots, s_n)$ by $true$.
   (c) replace $t \neq x$ by $x \neq t$ if t is not a variable.

   obtaining A',
4. if A' is a conjunction then return A'.
5. otherwise compute $dnf(A) = \vee(A_i)$ and return $\vee ResConj(A_i)$.

It is worth noting that the previous algorithm terminates since each constraint contains a finite number of inequalities.

Example 8 shows how the procedure Res computes the normal form of some constraints.

Example 8. 1. $c = (X = f(Y) \wedge (Y = a \vee Y = f(a)) \wedge \forall U\, X \neq f(f(U)))$.

$$\begin{aligned}
c_1 &= \forall U(\, X = f(Y) \wedge (Y = a \vee Y = f(a)) \wedge X \neq f(f(U))).\\
c_2 &= \forall U(\, (X = f(Y) \wedge Y \neq a \wedge Y = f(a) \wedge X \neq f(f(U)))\vee\\
&\qquad (X = f(Y) \wedge Y = a \wedge Y \neq f(a) \wedge X \neq f(f(U)))\vee\\
&\qquad (X = f(Y) \wedge Y = a \wedge Y = f(a) \wedge X \neq f(f(U)))).\\
c_{3.1} &= \forall U(\, (X = f(f(a)) \wedge Y \neq a \wedge Y = f(a) \wedge X \neq f(f(U)))\vee\\
&\qquad (X = f(a) \wedge Y = a \wedge Y \neq f(a) \wedge X \neq f(f(U))).\\
c_{3.2} &= \forall U(\, (X = f(f(a)) \wedge f(a) \neq a \wedge Y = f(a) \wedge f(f(a)) \neq f(f(U)))\vee\\
&\qquad (X = f(a) \wedge Y = a \wedge a \neq f(a) \wedge f(a) \neq f(f(U))).\\
c_{3.3} &= \forall U(\, (X = f(f(a)) \wedge Y = f(a) \wedge a \neq U)\vee\\
&\qquad (X = f(a) \wedge Y = a \wedge a \neq f(U))).\\
c_4 &= \qquad (X = f(a) \wedge Y = a).
\end{aligned}$$

2. $\bar{c} = (X = f(Z,S) \wedge U = (f(H),H) \wedge S = a \wedge X \neq U)$.

$$\begin{aligned}
c_2 &= c_1 = \bar{c}.\\
c_{3.1} &= (X = f(Z,a) \wedge U = f(f(H),H) \wedge S = a \wedge X \neq U).\\
c_{3.2} &= (X = f(Z,a) \wedge U = f(f(H),H) \wedge S = a \wedge f(Z,a) \neq f(f(H),H)).\\
c_{3.3} &= (X = f(Z,a) \wedge U = f(f(H),H) \wedge S = a \wedge (Z \neq f(H) \vee H \neq a).\\
c_{3.4} &= A_1 \vee A_2 \vee A_3.\\
A_1 &= (X = f(Z,a) \wedge U = f(f(H),H) \wedge S = a \wedge Z = f(H) \wedge H \neq a).\\
A_2 &= (X = f(Z,a) \wedge U = f(f(H),H) \wedge S = a \wedge Z \neq f(H) \wedge H \neq a).\\
A_3 &= (X = f(Z,a) \wedge U = f(f(H),H) \wedge S = a \wedge Z \neq f(H) \wedge H = a).
\end{aligned}$$

$$\mathit{ResConj}(A_1) = \bar{A_1} \;\; \mathit{ResConj}(A_2) = A_2 \;\; \mathit{ResConj}(A_3) = \bar{A_3}.$$

$$\begin{aligned}
\bar{A_1} &= (X = f(f(H),a) \wedge U = f(f(H),H) \wedge S = a \wedge Z = f(H) \wedge H \neq a).\\
\bar{A_3} &= (X = f(Z,a) \wedge U = f(f(a),a) \wedge S = a \wedge Z \neq f(a) \wedge H = a).\\
c_4 &= \bar{A_1} \vee A_2 \vee \bar{A_3}.
\end{aligned}$$

Note that all the steps in *ResConj* and *Res* preserve the $\mathcal{H}$-equivalence, the third step of *ResConj* is Colmerauer's simplification algorithm for inequalities [4], the first and second transformations are the usual ones for CET formulas [18], while the second step of *Res* is the extended disjunctive normal transformation [13]. Hence we get:

Proposition 9. $\mathcal{H} \models \phi \leftrightarrow \mathit{Res}(\phi)$.

Our concrete constraints domain $\mathcal{NC}$ will be the subset of constraints in $\mathcal{C}$ which are in *normal* form. The concrete operations on $\mathcal{NC}$ will be thus defined using the normal form:

Definition 10. Let $c_1, c_2 \in \mathcal{NC}$,
$c_1 \wedge^c c_2 = \mathit{Res}(c_1 \wedge c_2)$ $\quad c_1 \vee^c c_2 = \mathit{Res}(c_1 \vee c_2)$
$\neg^c c_1 = \mathit{Res}(\neg c_1)$ $\quad \exists^c X\; c_1 = \exists X\; c_1$ $\quad \forall^c X\; c_1 = \forall X\; c_1$

We denote by $\mathcal{B}$ the set of constrained atoms with constraints in $\mathcal{NC}$, and by $(\mathcal{I}, \subseteq)$ the complete lattice of (not necessarily consistent) partial constrained interpretations formed over $\mathcal{B}$.

4 Depth(k) analysis for constructive negation

The idea of depth(k) analysis was first introduced in [20]. The domain of depth(k) analysis was then used in order to approximate the ground success and failure sets for normal programs in [19].

We follow the formalization of [5] for positive logic programs. We want to approximate an infinite set of computed answer constraints by means of a constraint depth(k) cut, i.e. constraints where the equalities and inequalities are between variables and terms which have a depth not greater than k.

Our concrete domain is the complete lattice of partial constrained interpretations $(\mathcal{I}, \subseteq)$ of the previous section. Since our aim is to approximate the computed answer constraints, the fixpoint semantics we choose in the hierarchy [10] is the one which generalizes the S-semantics to normal logic programs, the $T_P^{\mathcal{B}_\mathcal{D}}$ operator (cf def. 1). The version we consider here is the one defined on the domain $\mathcal{B}$ with the concrete operations in $\mathcal{NC}$, $\wedge^c$, $\vee^c$, $\neg^c, \exists^c, \forall^c$, (the $T_P^{\mathcal{B}}$ operator).

4.1 The abstract domain

Terms are cut by replacing each-subterm rooted at depth greater than k by a new fresh variable taken from a set W, (disjoint from V). The depth(k) terms represent each term obtained by instantiating the variables of W with terms built over V.
Consider the depth function $||: Term \rightarrow Term$ such that

$$|t| = \begin{cases} 1 & \text{if } t \text{ is a constant or a variable} \\ max\{|t_1|, \ldots, |t_n|\} + 1 & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

and a given positive integer k. The abstract term $\alpha_k(t)$ is the term obtained from the concrete one by substituting a fresh variable (belonging to W) to each subterm t' in t, such that $|t| - |t'| = k$.
Consider now the abstract basic constraints

$$\mathcal{ABC} = \left\{ \begin{array}{l} c \mid \; c = (X = t) \;\; |t| \leq k \text{ or} \\ \quad\;\; c = (X' \neq t') \;\; |t'| \leq k, \text{ and } Var(t') \cap W = \emptyset \end{array} \right\}$$

Note that $Var(t') \cap W = \emptyset$ expresses the fact that inequalities between variables and cut terms are not allowed. The domain of abstract constraints is defined as follows,

Definition 11.

$$\mathcal{ANC} = \left\{ \begin{array}{l} c \mid \text{c is a constraint in normal form built with} \\ \quad\;\; \text{the logical connectives } \vee, \wedge, \forall \text{ and } \exists \text{ on } \mathcal{ABC} \end{array} \right\}$$

The concepts of abstract constrained atoms and partial abstract interpretations are defined as expected.

Definition 12. An abstract constrained atom is a pair $c|A$ such that $c \in \mathcal{ANC}$ and c is a $\mathcal{H} - solvable$ constraint, A is an atom and $Var(c) \subseteq Var(A)$. With $\mathcal{B}^a$ we intend the set of abstract constrained atoms.

The abstract domain is the set of partial interpretations on abstract constrained atoms. A *partial abstract constrained interpretation* for a program, is a pair of set of abstract constrained atoms, $I^a =< I^{a^+}, I^{a^-} >$, not necessary consistent. We consider $\mathcal{I}^a = \{I^a | \; I^a \text{ is a partial interpretation}\}$.

With respect to the case of definite logic programs [5], we need to define a different order on the abstract constraint domain.

This is because the result c^a of an abstract *and* operation on the abstract constraint domain will be an approximation of the result c of the concrete *and* operation on the concrete constraint domain, in the sense that c^a will be "more general" than the abstraction of c (where here "more general" means "is implied under $\mathcal{H}$") .
This motivates the definition of the following relation on the abstract constraint domain.

Definition 13. Let $c, c' \in \mathcal{ANC}$. $c \preceq_a c'$ if $\mathcal{H} \models c \rightarrow c'$.

We consider the order $\leq_a$ induced by the preorder $\preceq_a$, namely the order obtained considering the classes modulo the equivalence induced by $\preceq_a$.
We define the downward closure of a pair of sets w.r.t. the $\leq_a$ order,

Definition 14. Consider a pair of sets of constrained atoms B.
By $\downarrow B$ we denote the downward closure of $< B^+, B^- >$.
$c|A \in \downarrow B^+$ *if there exists* $c'|A \in B^+$ and $c \leq_a c'$,
$c|A \in \downarrow B^-$ *if there exists* $c'|A \in B^-$ and $c \leq_a c'$.

Intuitively, a set of constrained atoms I is less or equal than J, if $\downarrow I \subseteq \downarrow J$.

Definition 15. Consider $I, J \in \mathcal{I}^a$.

$$\begin{aligned} I^a \preceq J^a \leftrightarrow & \text{ for all } c|A \in {I^a}^+ \;\; \exists c'|A \in {J^a}^+ \text{ such that } c \leq_a c' \text{ and} \\ & \text{ for all } c|A \in {I^a}^- \;\; \exists c'|A \in {J^a}^- \text{ such that } c \leq_a c' \end{aligned}$$

It is immediate to see that $\preceq$ defines a preorder. We consider the order $\leq$ induced by the preorder $\preceq$, namely the order obtained by considering the classes $\mathbb{I}^a$ modulo the equivalence induced by $\preceq$. Then our abstract domain is $(\mathbb{I}^a, \leq)$. Since the operations on the equivalence classes are independent on the choice of the representative, we denote the class of an interpretation I^a by I^a itself. In the rest of the paper, we often abuse notation by denoting by I^a the equivalence class of I^a or the interpretation I^a itself.

4.2 The abstraction function

Let us now define the abstraction function. To this aim we first define the function α_c on constraints. The main idea is to define α_c on the basic constraints as follows: an equality $X = t$ is abstracted to $X = \alpha_k(t)$, while an inequality $X \neq t$ is abstracted to $X \neq t$ if $|t| \leq k$ and to *true* otherwise.
We denote by $\Delta(c)$ the constraint c' in normal form and by Δ the sequence of quantified variables of c', where c is the quantifier-free part of c'.

Definition 16. The depth(k) constraint abstraction function is the function $\alpha_c : \mathcal{NC} \rightarrow \mathcal{ANC}$:
$\alpha_c(\Delta(c)) = \Delta, \Delta' \alpha_c(c)$ where $\Delta' = \exists Y_1, \exists Y_2, .., $ and $Y_i \in (W \cap Var(\alpha_c(c)))$
$\alpha_c(X = t) = (X = \alpha_k(t))$,
$\alpha_c(false) = false$, $\qquad \alpha_c(true) = true$,
$\alpha_c(X \neq t) = (X \neq t)$ if $|t| \leq k$, $\qquad \alpha_c(X \neq t) = true$ if $|t| > k$,
$\alpha_c(A \wedge B) = \alpha_c(A) \wedge \alpha_c(B)$, $\qquad \alpha_c(A \vee B) = \alpha_c(A) \vee \alpha_c(B)$.

Note that the first definition means that all the new variables introduced by the cut terms have to be considered existentially quantified.
Example 17 shows an application of α_c.

Example 17. $c = \forall U((H = f(f(T)) \wedge T \neq f(f(U)) \wedge X = f(U)) \vee$
$(H = f(f(T)) \wedge T \neq f(X) \wedge X \neq f(U))),\ k = 2.$

$$\begin{array}{lll}
\alpha_c(c) = \alpha_c(\ \forall U(& & (H = f(f(T)) \wedge T \neq f(f(U)) \wedge X = f(U)) \vee \\
 & & (H = f(f(T)) \wedge T \neq f(X) \wedge X \neq f(U)))) = \\
 & \forall U(& (\alpha_c(H = f(f(T))) \wedge \alpha_c(T \neq f(f(U))) \wedge \alpha_c(X = f(U))) \vee \\
 & & (\alpha_c(H = f(f(T))) \wedge \alpha_c(T \neq f(X)) \wedge \alpha_c(X \neq f(U))) = \\
 & \forall U, \exists Q_1, Q_2 & ((H = f(Q_1) \wedge \mathit{true} \wedge X = f(U)) \vee \\
 & & (H = f(Q_2) \wedge T \neq f(X) \wedge X \neq f(U))) \quad (Q_1, Q_2 \in W).
\end{array}$$

The abstraction function α is defined by applying α_c to every constraint of the constrained atoms in the concrete interpretation.

Definition 18. Let $\alpha : \mathcal{I} \to \mathbb{I}^a$: $\alpha = < \alpha^+, \alpha^- >$
$\alpha^+(I) = \{c|A \mid\ c'|A \in I^+ \text{ and } \alpha_c(c') = c\}$,
$\alpha^-(I) = \{c|A \mid\ c'|A \in I^- \text{ and } \alpha_c(c') = c\}$.

As a consequence the function γ on (equivalence classes of) sets of abstract constraints is automatically determined as follows:

Definition 19. Let $\gamma : \mathbb{I}^a \to \mathcal{I}$:

$$\begin{array}{l}
\gamma(I^a) = \cup\{I \mid \alpha(I) \leq I^a\} = \\
\qquad \cup\{I \mid \forall c|A \in \alpha^+(I)\ \exists c'|A \in {I^a}^+ \text{ such that } c \leq_a c' \text{ and} \\
\qquad\qquad \forall c|A \in \alpha^-(I)\ \exists c'|A \in {I^a}^- \text{ such that } c \leq_a c'\} = \\
\qquad \cup\{I \mid \downarrow \alpha(I) \subseteq \downarrow I^a\} = \ \cup\{I \mid \alpha(I) \subseteq \downarrow I^a\}
\end{array}$$

Lemma 20. *α is additive.*

Theorem 21. *$< \alpha, \gamma >$ is a Galois insertion of $(\mathcal{I}, \subseteq)$ into $(\mathbb{I}^a, \leq)$.*

4.3 α_c is a congruence w.r.t. the H-equivalence

As we have already pointed out in section 3, we want to work with $\mathcal{H}$-equivalence classes of constraints and, for this purpose, we need to be sure that the above defined function α_c on $\mathcal{NC}$ is a congruence w.r.t. the $\mathcal{H}$-equivalence. This means that if two constraints $c, c' \in \mathcal{NC}$ are $\mathcal{H}$-equivalent, then also $\alpha_c(c)$ and $\alpha_c(c')$ have to be $\mathcal{H}$-equivalent.

In order to understand whether two constraints are $\mathcal{H}$-equivalent, it is useful to state the following result.

Lemma 22. *Consider the inequality $X \neq t$. There exist no arbitrary quantified $t_1, \ldots, t_n$, where $t_i \neq t$, such that $X \neq t$ is $\mathcal{H}$-equivalent to $\wedge_i X \neq t_i$.*

This is a consequence of the fact that we consider the models of the theory CET without the DCA axiom.

The previous result, together with the fact that constraints are in false-simplified form, allows us to claim that α_c is a congruence.

Theorem 23. *Let $c, c' \in \mathcal{NC}$. If $\mathcal{H} \models c \leftrightarrow c'$ then $\mathcal{H} \models \alpha_c(c) \leftrightarrow \alpha_c(c')$.*

4.4 The abstract fixpoint operator

We now define the abstract operations that will replace the concrete ones in the definition of the fixpoint abstract operator. We show that the abstract operations are a correct approximation of the concrete operations.
The definition of the abstract *and* operation is not immediate. The example 24 is meant to give some intuition on some problems that may arise.

Example 24. Consider the following two constraints:
$c_1 = (X = f(Z, f(H)) \wedge S = f(a))$ $c_2 = (U \neq X \wedge Y \neq f(S))$ and $k = 2$.
Consider $\alpha_c(c_1) = \exists Q(X = f(Z, Q) \wedge S = f(a))$ $\alpha_c(c_2) = (U \neq X \wedge Y \neq f(S))$.
If we now consider the normalized form of $\alpha_c(c_1) \wedge \alpha_c(c_2)$ the resulting constraint is $\exists Q(U \neq f(Z, Q) \wedge Y \neq f(f(a)) \wedge X = f(Z, Q) \wedge S = f(a))$, which is not an abstract constraint according to definition 11.

The problem is that the normalized form of the logical *and* operation on two abstract constraints is not in general an abstract constraint (the depth of the terms involved in equalities and inequalities can be greater than k and it can contain inequalities between variables and cut terms).
This is the reason why we need to define a new $\mathcal{M}$ operator, on the normalized forms of abstract constraints. The $\mathcal{M}$ operator must cut terms deeper than k and replace by true all the inequalities which contain a cut term. Intuitively this is because $X \neq t$, where $Var(t) \cap W \neq \emptyset$, represents, on the concrete domain, an inequality between a variable and a term longer than k. On the abstract domain, such inequalities are abstracted to the constant *true*.

Definition 25. Let $\mathcal{M} : \mathcal{NC} \to \mathcal{ANC}$

$\mathcal{M}(\Delta(c)) = \Delta, \Delta'\mathcal{M}(c)$ where $\Delta' = \exists Y_1, \exists Y_2, ..,$ where $Y_i \in (W \cap Var(\mathcal{M}(c)))$.
$\mathcal{M}(X = t) = (X = \alpha_k(t))$
$\mathcal{M}(X \neq t) = (X \neq t)$ if $|t| \leq k$ and $Var(t) \cap W = \emptyset$
$\mathcal{M}(X \neq t) = (true)$ if $|t| > k$ or $Var(t) \cap W \neq \emptyset$
$\mathcal{M}(A \wedge B) = \alpha_c(A) \wedge \alpha_c(B)$, $\quad \mathcal{M}(A \vee B) = \alpha_c(A) \vee \alpha_c(B)$

As expected, the $\mathcal{M}$ operator is similar to the α_c operator. The only difference is that $\mathcal{M}$ replace by *true* all the inequalities between variables and cut terms. Since $\mathcal{ANC}$ is a subset of $\mathcal{NC}$, the *Res* form is defined also on the abstract constraints domain.

Definition 26. Let $c_1, c_2 \in \mathcal{ANC}$
$c_1 \tilde{\wedge} c_2 = \mathcal{M}(Res(c_1 \wedge c_2))$, $\quad c_1 \tilde{\vee} c_2 = \mathcal{M}(Res(c_1 \vee c_2))$,
$\tilde{\neg} c_1 = \mathcal{M}(Res(\neg c_1))$, $\quad \tilde{\exists} X\ c_1 = \exists X\ c_1$, $\quad \tilde{\forall} X\ c_1 = \forall X\ c_1$,

It is worth noting that the procedure *Res* on the abstract domain needs to perform the logical *and* on abstract constraints. This means that most of the observations that can be done on the behavior of the abstract *and* operation, concern also the abstract *or* and *not* operations.

Example 27 illustrates the relation between the abstract *and* operation and the abstraction of the concrete *and* operation. For a sake of simplicity, since in this case it does not affect the result, we write the constraint c_1 in the more compact standard disjunctive form rather than of in extended disjunctive form.

Example 27. $c_1 = \forall K((Y = a \wedge U \neq f(f(K))) \vee Z = a)$, $c_2 = (U = f(f(a)))$.
Consider $k = 1$. $\alpha_c(c_1) = (Y = a \vee Z = a)$, $\alpha_c(c_2) = \exists V\ U = f(V)$.
$\alpha_c(c_1)\tilde{\wedge}\alpha_c(c_2) = \exists V((Y = a \wedge U = f(V)) \vee (Z = a \wedge U = f(V)))$.
$\alpha_c(Res(c_1 \wedge c_2)) = \exists V(Z = a \wedge U = f(V))$.
$\mathcal{H} \models \alpha_c(Res(c_1 \wedge c_2)) \rightarrow \alpha_c(c_1)\tilde{\wedge}\alpha_c(c_2)$

As already pointed out, the abstract *and* gives a more general constraint than the abstraction of the one computed by the concrete *and* and this is the reason why we have defined an approximation order based on implication (under $\mathcal{H}$) between constraints.

In order to show that the abstract operations are correct, we prove a stronger property.

Theorem 28. *Let* $c_1, c_2 \in \mathcal{NC}$.
$\alpha_c(c_1)\tilde{\wedge}\alpha_c(c_2) \geq_a \alpha_c(c_1 \wedge^c c_2)$, $\quad \alpha_c(c_1)\tilde{\vee}\alpha_c(c_2) \geq_a \alpha_c(c_1 \vee^c c_2)$,
$\tilde{\exists} x\ \alpha_c(c_1) = \alpha_c(\exists^c x\ c_1)$, $\quad \tilde{\forall} x\ \alpha_c(c_1) = \alpha_c(\forall^c x\ c_1)$.

As shown by example 29, the correctness property does not hold for the version of abstract "not" which we have defined, if we consider general constraints.

Example 29. Consider $c_1 = (X \neq f(f(a)))$ and $k = 1$.
$\alpha_c(\neg^c(c_1)) = \exists Y\ X = f(Y)$ which does not implies $\tilde{\neg}(\alpha_c(c_1)) = false$.

Since the *not* operator is used by the abstract fixpoint operator on "simpler " constraints (the program constraints) only, we are interested in its behavior on conjunctions of equalities between variables and terms only. For this kind of constraints the following result holds.

Lemma 30. *If* $c_1 = (\bigwedge_i (X_i = t_i)) \in \mathcal{NC}$, *then* $\tilde{\neg}\alpha_c(c_1) \geq_a \alpha_c(\neg^c(c_1))$.

Now that we have defined the abstract constraints domain and the abstract operations, we can define the abstract fixpoint operator.

Definition 31. Let $\alpha(P)$ be the program obtained by replacing every constraint c in a clause of P by $\alpha_c(c)$.
The abstract fixpoint operator: $\mathbb{I}^a \rightarrow \mathbb{I}^a$ is defined as follows, $T_P^{\mathcal{B}^a}(I^a) = T_{\alpha(P)}^{\mathcal{B}^a}(\downarrow I^a)$, where the operations are $\tilde{\exists}$, $\tilde{\forall}$, $\tilde{\neg}$ on $\mathcal{ANC}$ and $\tilde{\vee}$, $\tilde{\wedge}$ on $\mathcal{ANC} \times \mathcal{ANC}$.

By definition, $T_P^{\mathcal{B}^a}$ is a congruence respect to the equivalence classes of the abstract domain. Note also that $T_P^{\mathcal{B}^a}$ is monotonic on the $(\mathbb{I}^a, \leq)$, because $I \leq J$ implies $\downarrow I \subseteq \downarrow J$.

Lemma 32. $T_P^{\mathcal{B}^a}$ *is monotonic on the* $(\mathbb{I}^a, \leq)$.

The proof that the abstract operator is correct w.r.t. the concrete one, is based on the correctness of the abstract operations on the abstract constraints domain.

Theorem 33. $\alpha(T_P^{\mathcal{B}}(\gamma(I^a))) \leq T_P^{\mathcal{B}^a}(I^a)$. *Then* $\alpha(lfp(T_P^{\mathcal{B}})) \leq T_P^{\mathcal{B}^a}(I^a)$.

Consider now a k greater than the maximal depth of the terms involved in the constraints of the clauses in the program P. In this case the abstract operator is also optimal.

Theorem 34. $T_P^{\mathcal{B}^a}(I^a) \leq \alpha(T_P^{\mathcal{B}}(\gamma(I^a)))$.

Let us finally discuss termination properties of the dataflow analyses presented in this section. First note that the set of not equivalent (w.r.t. $\mathcal{H}$) set of constraints belonging to $\mathcal{ANC}$ is finite.

Lemma 35. *Assume that the signature of the program has a finite number of function and predicate symbols. Our depth(k) abstraction is ascending chain finite.*

4.5 An example

We now show how the depth-k analysis works on an example.The program of figure 1 computes the union of two sets represented as lists. We denote the equivalence class of $T_P^{\mathcal{B}^a}$ by $T_P^{\mathcal{B}^a}$ itself. All the computed constraints for the predicate $\neg member$ are shown, while concerning the predicate $\neg union$, for a sake of simplicity, we choose to show just a small subset of the computed answer constraints (written in the more compact standard disjunctive form). Therefore, the concretization of the set of answer constraints for $\neg union$ that we present in figure 1, contains some answer constraints computed by the concrete semantics but not all of them.

As expected the set of answer constraints, computed by the abstract fixpoint operator, is an approximation of the answer constraints, computed by the concrete operator, for the predicates *member*, *union* and $\neg member$. For example, for the predicate $\neg member(X, Y)$, we compute the answer $\forall L(Y \neq [X, L])$ which correctly approximates the concrete answer $\forall L, H, H_1, L_1(Y \neq [X, L] \wedge Y \neq [H, H_1, L_1])$. While the constraint answer $\exists X \forall H_1, L_1 \exists Z_1, Z_2(A = [X, Z_1] \wedge C = [X, Z_2] \wedge B \neq [H_1, L_1])$ for $union(A, B, C)$, approximates the concrete constraint $A = [X, X]$, $C = [X, X, K]$, $B = K$ and B is not a list, computed by the concrete semantics. Note, in fact, that, if the second argument is not a list, the predicate *member* finitely fails. Let us now consider Marriott and Søndergaard's abstraction for the program P, with a language where the only constant is a (this assumption does not affect the result). Concerning the predicate *union* with the empty list as first argument, their abstraction computes the following atoms $union([\ ], a, a)$, $union([\ ], [\ ], [\ ])$, $union([\ ], [a], [a])$, $union([\ ], [a, Z_1], [a, Z_2])$, while we obtain the more precise answer $(A = [\ ] \wedge B = C) | union(A, B, C)$.

$P:$
$union(A,B,C):-A=[\,],B=C.$
$union(A,B,C):-A=[X,L],C=[X,K],\neg member(X,B),union(L,B,K).$
$union(A,B,C):-A=[X,L],member(X,B),union(L,B,C).$
$member(X,Y):-Y=[X,L].$
$member(X,Y):-Y=[H,L],member(X,L).$

Consider now a depth-2 analysis with $Z_i \in W$.

$$T_P^{\mathcal{B}^{a+}}$$

$$\exists L(\quad Y=[X,L] \quad)|member(X,Y).$$

$$\exists H,Z_1(\quad Y=[H,Z_1] \quad)|member(X,Y).$$

$$A=[\,]\wedge B=C \quad |union(A,B,C).$$

$$\exists X,L(\quad A=[X]\wedge B=[X,L]\wedge B=C \quad)|union(A,B,C).$$

$$\exists X,Y,Z_1(\quad A=[X]\wedge B=[Y,Z_1]\wedge B=C \quad)|union(A,B,C).$$

$$\exists X,H,L,Z_1(\quad A=[X,Z_1]\wedge B=[X,L]\wedge B=C \quad)|union(A,B,C).$$

$$\exists X,H,Z_1,Z_2(\quad A=[X,Z_1]\wedge B=[H,Z_2]\wedge B=C \quad)|union(A,B,C).$$

$$\exists X,K\forall H,L(\ A=[X]\wedge C=[X,K]\wedge B\neq[H,L]\wedge B=K \ )|union(A,B,C).$$

$$\exists X\forall H,L\exists Z_1,Z_2(\quad A=[X,Z_1]\wedge C=[X,Z_2]\wedge B\neq[H,L] \quad)|union(A,B,C).$$

$$\exists X,K\forall L(\ A=[X]\wedge C=[X,K]\wedge B\neq[X,L]\wedge B=K \ )|union(A,B,C).$$

$$\exists X\forall L\exists Z_1,Z_2(\quad A=[X,Z_1]\wedge C=[X,Z_2]\wedge B\neq[X,L] \quad)|union(A,B,C).$$

A subset of $T_P^{\mathcal{B}^{a-}}$ (complete for the predicate *member*)

$$\forall H,L(\quad Y\neq[H,L] \quad)|member(X,Y).$$

$$\forall L(\quad Y\neq[X,L] \quad)|member(X,Y).$$

$$\begin{array}{ll} \forall X,L,KX_1,L_1 & ((A\neq[\,]\wedge A\neq[X,L])\vee \\ & (B\neq C\wedge A\neq[X,L]\vee \\ & (B\neq C\wedge C\neq[X,K])\wedge A\neq[X_1,L_1]) \quad)|union(A,B,C). \end{array}$$

$$\begin{array}{ll} \forall X,L,KX_1,L_1,H,L_2 & ((A\neq[\,]\wedge A\neq[X,L])\vee \\ & (B\neq C\wedge A\neq[X,L]\vee \\ & (B\neq C\wedge C\neq[X,K])\wedge A\neq[X_1,L_1])\vee \\ & (B\neq C\wedge C\neq[X,L]\wedge B\neq[H,L_2])\vee \\ & (A\neq[\,]\wedge C\neq[X,L]\wedge B\neq[H,L_2]) \quad |)union(A,B,C) \end{array}$$

$$\vdots$$

Fig. 1. Example 1

The atom $union([\ ], [a, Z_1], [a, Z_2])$, in fact, correctly approximates the predicates deeper than k which have a successful behavior, but it has lost the relation between B and C. As a consequence all the other ground atoms for *union* computed using the atom $union([\ ], [a, Z_1], [a, Z_2])$, are less precise than the ground instances of the atoms computed by our non-ground abstract semantics.

5 Conclusion

Starting from the hierarchy of semantics defined in [10], our aim was to show that well known analysis for logic programs could be extended to normal logic programs. Based on the framework of abstract interpretation [7,8], we have presented a depth(k) analysis which is able to approximate the answer set of normal logic programs.

It is worth noting that our depth(k) analysis, can be easily generalized to constraint logic programs defined on $\mathcal{H}$, whose program constraints can be conjunctions of equalities and inequalities. In order to deal with constructive negation, in fact, most of the results presented in this paper hold for first order equality constraints. The only exception is lemma 30 (and consequently theorem 33 and theorem 34), which is true only for conjunctions of equalities. But a more complex definition of the abstract *not* operator can be defined and proven correct on conjunctions of equalities and inequalities constraints. This alternative definition is, however, less precise than the one defined here. As a consequence theorem 33, where the abstract fixpoint operator uses the new abstract *not* operator, still holds for such "extended" logic programs, while it is not the case for theorem 34.

References

1. K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier and The MIT Press, 1990.
2. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19–20:149–197, 1994.
3. D. Chan. Constructive Negation Based on the Completed Database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 111–125. The MIT Press, 1988.
4. A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer System*, pages 85–99, 1984.
5. M. Comini. *Abstract Interpretation framework for Semantics and Diagnosis of Logic Programs.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1998.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

7. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.
8. P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
9. F. Fages. Constructive negation by pruning. *Journal of Logic Programming*, 32(2):85–118, 1997.
10. F. Fages and R. Gori. A hierarchy of semantics for normal constraint logic programs. In M.Hanus M.Rodriguez-Artalejo, editor, *Proc. Fifth Int'l Conf. on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 1996.
11. M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2:295–312, 1985.
12. R. Giacobazzi. "Optimal" collecting semantics for analysis in a hierarchy of logic program semantics. In C. Puech and R. Reischuk, editors, *Proc. 13th International Symposium on Theoretical Aspects of Computer Science (STACS'96)*, volume 1046 of *Lecture Notes in Computer Science*, pages 503–514. Springer-Verlag, 1996.
13. S. Koppelberg. *Handbook of Boolean Algebras (Vol.I)*. Elsevier Science Publisher B.V.(North Holland), 1989.
14. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.
15. K. Kunen. Signed Data Dependencies in Logic Programs. *Journal of Logic Programming*, 7(3):231–245, 1989.
16. J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.
17. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. Second edition.
18. M.J. Maher. Complete axiomatizations of the algebra of finite, rational and infinite trees. In *Third Symp. on Logic in Computer Science*, pages 348–357, 1988.
19. K. Marriott and H. Sondergaard. Bottom-up Dataflow Analysis of Normal Logic Programs. *Journal of Logic Programming*, 13(2 & 3):181–204, 1992.
20. T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.
21. P. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.