Widening Pos for Efficient and Scalable Groundness Analysis of Logic Programs

Florence Benoy[1], **Mike Codish**[2], **Andy Heaton**[3], and **Andy King**[1]
[1]University of Kent, CT2 7NF, UK. amk@ukc.ac.uk
[2]Ben-Gurion University, PoB 653, Israel. codish@cs.bgu.ac.il
[3]University of Leeds, LS2 9JT, UK. heaton@scs.leeds.ac.uk

Abstract
Positive (Pos) and definite (Def) Boolean functions have been successfully applied to infer dependencies between the variables of a logic program. Two new analysis techniques for Boolean functions are presented. The first is an easy to implement, fast and scalable technique for the goal dependent analysis of logic programs. It is based on widening Pos. The second technique is based on using linear constraints to represent formulae in Def. The resulting analysis is both easy to implement and precise. It is more suited to the analysis of constraint programs.

1 Introduction

Many analyses for logic programs use Boolean functions to express dependencies between program variables. The most commonly used are the classes of positive and definite propositional formula, denoted *Pos* and *Def*, respectively [1]. Numerous independent implementations have indicated that program analyses based on *Pos* [1, 2, 3, 6, 9, 10, 13, 14] and *Def* [15] formula are accurate and well suited to the analysis of logic programs, constraint logic programs and deductive databases. One of the best known applications of this type of analysis is in reasoning about groundness dependencies. For example, in this context the definite formula $x \wedge (y \leftarrow z)$ is interpreted to describe a program store in which x is definitely ground and there exists a grounding dependency such that whenever z becomes ground then so does y.

A variety of techniques have been devised to represent, maintain and manipulate various classes of propositional formula for program analyses. Worth mentioning are the use of binary decision diagrams (BDD) and their variants, like reduced order binary decision diagrams (ROBDD) [2, 3, 10], which lead to fast analyses for many benchmarks. In this approach the representation of a formula is concise. What is not clear, however, is whether the representation is always small enough to guarantee that the analysis will scale for programs with clauses which contain large numbers of variables. BDD and ROBDD based analysers, or at least their domain operations, are typically implemented in C [1, 2, 9].

On the other hand, the Prolog based implementation technique of [6], in which propositional formulae are represented as sets of atoms, suffers from a fundamental scaling problem. In this approach a set of atoms for a predicate p can be viewed as a representation of the disjunctive normal form of the formula associated with p. The advantage of this scheme is that an implementation can be based on the underlying backtracking and unification operations provided by Prolog. The disadvantage is that the analysis is non-deterministic in the sense that it proceeds by iteratively matching the atoms in each clause body with the atoms describing the corresponding predicates. The problem is that, in the worst case, a predicate of arity n may be described by 2^n atoms. Thus the analysis does not always scale well for large programs.

This paper plugs two gaps. First, in section 3, we explain how the scaling problem of [6] can be finessed for query-dependent analysis by storing at most *one* (possibly non-ground) atom per predicate. Formally, this approximation corresponds to a widening [11]. The surprising result is that for a wide range of Prolog programs, this approximation does not significantly effect the amount of groundness information obtained in either the call patterns or the answer patterns! We built on this finding and go on to describe a practical and very simple Prolog based groundness dependency analysis for Prolog programs which is robust enough to analyse large programs with almost no loss of precision. Some experimental results for goal-dependent analysis are presented.

Our second contribution relates to the groundness (or rather definiteness [3, 15]) analysis of constraint logic programs. The form of dependencies that arise in constraint programs differ from those that occur in logic programs. For example, the Herbrand constraint $x = f(y, z)$ is described by $x \leftrightarrow (y \wedge z)$ – a bidirectional implication, whereas the linear constraint, $x = y + z$ is described by $(x \leftarrow (y \wedge z)) \wedge (y \leftarrow (x \wedge z)) \wedge (z \leftarrow (x \wedge y))$ – a conjunction of implications. Our groundness analysis, however, cannot always propagate these sophisticated dependencies. It is also imprecise for goal-independent program analysis. Section 4 addresses this problem by exploiting a connection between *Def* and the domain of polyhedral convex sets [12, 19]. Another surprising result is that the the meet operation (conjunction) in *Def* can be modeled by the conjunction of linear inequalities whereas the join operation in *Def* (an upper approximation of classical disjunction [1]) can be modeled by the convex hull of polyhedra. Since the convex hull operation can be implemented straightforwardly in a language with linear constraints [4] this leads to another simple-to-implement, but precise, analysis. Since the analysis only requires support for linear constraints, it gives a way implementing a definiteness analysis in a system like CLP($\mathcal{R}$). The scaling behaviour of the analysis is demonstrated for goal-independent analysis.

The rest of the paper is structured as follows. Section 2 surveys the necessary preliminaries; section 5 reviews the related work; section 6 discusses some directions for future work; and finally section 7 presents some concluding remarks.

2 Preliminaries

2.1 Positive and definite Boolean functions

We say that a variable set M is a model for a Boolean function f if $(\wedge M) \models f$. We let $model_X(f)$ denote the set of models of a Boolean function f over a set of variables X, that is, $model_X(f) = \{M \subseteq X \mid (\wedge M) \models f\}$. Hence, if $X = \{x, y\}$ then $model_X(x \wedge y) = \{\{x, y\}\}$ and $model_X(x \vee y) = \{\{x\}, \{y\}, \{x, y\}\}$. Pos_X denotes the set of positive Boolean functions whereas Def_X denotes the functions of Pos_X that are also definite. Recall that a Boolean function f is positive if $X \in model_X(f)$ and definite if $model_X(f)$ is closed under intersection, that is, $M \cap M' \in model_X(f)$ for all $M, M' \in model_X(f)$. Hence $x \wedge y$, $x \vee y \in Pos_X$ but only $x \wedge y \in Def_X$ since

$\{x\}, \{y\} \in model_X(x \vee y)$ and $\emptyset \notin model_X(x \vee y)$. A definite function over X can always be represented as a formula of the form $\wedge_{i=1}^{n}(x_i \leftarrow \wedge X_i))$ where $x_i \in X$ and $X_i \subseteq X$ [1].

2.2 Abstracting constraints with Boolean functions

The abstract domain Pos_X is formally defined in [8, 18] as is the precise relationship between concrete data, typically Herbrand constraints, and the Boolean functions that represent them. When Boolean functions are interpreted as describing groundness dependencies, the formula x expresses that the program variable x is (definitely) bound to a ground term. The formula $x \leftrightarrow y$, on the other hand, expresses the fact that x is bound to a ground term if and only if y is, so an eventual binding of x to a ground term will be propagated to y and *vice versa*. The interested reader can find additional details on analyses with Pos_X in [6, 9, 14] where bottom-up evaluation and top-down evaluation with tabling are considered.

2.3 Abstracting programs with Boolean functions

Let us first recall briefly the implementation technique for groundness dependencies that is described in [6]. In this approach, a given logic program is first abstracted in such a way that the concrete semantics of the abstract program corresponds to the required groundness analysis. Figure 1 illustrates a Prolog program which rotates the elements of a list (on the left) and its corresponding abstraction (on the right) for groundness dependencies. The predicate $\mathit{iff}(x, x_1, x_2)$ specifies the formula $x \leftrightarrow x_1 \wedge x_2$ with the intended interpretation that x is ground if and only if x_1 and x_2 are. Consequently, a unification of the form $x = [y|z]$ in the concrete program is replaced by a literal $\mathit{iff}(x, y, z)$. Similarly, a unification of the form $x = [\,]$ in the concrete program can simply be replaced by $\mathit{iff}(x)$ which specifies that x is definitely bound to a ground term. For the purpose of this paper it is sufficient to understand that the problem of analyzing the concrete program (on the left part of Figure 1) has been reduced to the problem of applying concrete evaluation to the abstract program (on the right). For additional details on why this is so, refer to [6, 8, 18].

Concrete rotate	Abstract rotate & Auxiliary predicates

<pre>rotate(Xs,Ys) :-
    append(As,Bs,Xs),
    append(Bs,As,Ys).
 	
append(Xs,Ys,Zs) :-
    Xs = [ ],
    Ys = Zs.
append(Xs,Ys,Zs) :-
    Xs = [X|Xs1],
    Zs = [X|Zs1],
    append(Xs1,Ys,Zs1).</pre>
	<pre>rotate(Xs,Ys) :- iff(true).
	    append(As,Bs,Xs),
	    append(Bs,As,Ys). iff(true,true).
	                         iff(false,false).
	append(Xs,Ys,Zs) :-
	    iff(Xs), iff(true,true,true).
	    iff(Ys,Zs). iff(false,true,false).
	append(Xs,Ys,Zs) :- iff(false,false,true).
	    iff(Xs,X,Xs1), iff(false,false,false).
	    iff(Zs,X,Zs1),
	    append(Xs1,Ys,Zs1).</pre>

Figure 1: Corresponding concrete and abstract programs.

2.4 Simple interpreter for bottom-up analysis

The scaling problems that arise in the analysis of [6] stem from the evaluation strategy that is used within the analyser. Figure 2 illustrates a simple Prolog interpreter (and thus an analyser) for naive bottom-up evaluation of any (finitary) logic program P. Each clause $h \leftarrow b_1, \ldots, b_n$ in P is represented as a fact of the form $user\_clause(h,\ [b_1, \ldots, b_n])$. The interpreter can be divided conceptually into two components. On the right, the predicate `operator/0` provides the "logic" and the inner loop of the algorithm which for each $user\_clause(Head, Body)$ in P proves the $Body$ using facts derived so far and calls the predicate `cond_assert/1` which asserts the $Head$ if it is new. A fact is new if it is not subsumed by any of the facts derived so far. The iff/n predicates introduced by the abstraction are solved as builtins using the second clause of $prove/1$. The control component, on the left, invokes iterations of the `operator` until no new facts are derived. When a new fact is asserted to the Prolog database, a *flag* is raised (unless the flag has already been raised). Iteration terminates when $retract(flag)$ fails in the second clause indicating that no new facts were asserted in the previous iteration. Bottom-up evaluation is initiated by the query `?- iterate` which leaves the result of the evaluation in the Prolog database. This provides for goal independent analyses. Goal dependent analyses are supported by applying Magic sets or similar techniques. For further details see [6]. The interpreter can be used as the basis for various more efficient implementations based on semi-naive evaluation, strongly connected components, and other optimization techniques.

$$
\begin{array}{l}
iterate \leftarrow operator,\ fail.\\
iterate \leftarrow retract(flag),\ iterate.\\
iterate.\\[4pt]
cond\_assert(F) \leftarrow in\_database(F),\ !.\\
cond\_assert(F) \leftarrow assert(F),\\
\quad cond\_assert(flag).\\[4pt]
in\_database(G) \leftarrow functor(G, N, A),\\
\quad functor(B, N, A),\ call(B),\\
\quad subsumes(B, G),\ !.
\end{array}
$$

The "control"

$$
\begin{array}{l}
operator \leftarrow\\
\quad user\_clause(Head, Body),\\
\quad prove(Body),\\
\quad cond\_assert(fact(Head)).\\[4pt]
prove([\ ]).\\
prove([B|Bs]) \leftarrow\\
\quad builtin(B), !,\\
\quad call(B), prove(Bs).\\
prove([B|Bs]) \leftarrow\\
\quad fact(B),\ prove(Bs).
\end{array}
$$

The "logic"

Figure 2: A Prolog interpreter for bottom-up evaluation.

3 Scalable groundness analysis by widening Pos

The main problem in basing analyses on the approach depicted in Figure 2 is the non-determinism introduced in solving the body of a clause (predicate $prove/2$) using the facts derived so far. In general, for each predicate p/n defined in the program, up to 2^n atoms may be needed to represent its groundness dependencies. Using a non-ground representation has some advantages. For example, the single non-ground atom $q(x, x)$ represents the two ground atoms $q(true, true)$, $q(false, false)$, and the three non-ground atoms $p(true, true, true)$, $p(false, \_, false)$, $p(false, false, \_)$

represent the atoms $p(true, true, true)$, $p(false, true, false)$, $p(false, false, true)$ and $p(false, false, false)$. Indeed, dependency information for a predicate with two or three arguments can always be stored using at most two or three atoms respectively. It is interesting to note that the non-ground representation often reduces considerably the size of a formula. For example, a formula on n variables represented by all of the 2^n ground atoms for p/n can also be represented by the single atom $p(\_, \ldots, \_)$. Introducing a single variable as an argument in the atoms of a ground representation can half the size of the representation. However, as the number of variables increases, the numbers of non-ground atoms required to represent arbitrary, positive and definite propositional formula still increases exponentially. In this section we investigate three techniques in which the size of the representation of a formula can be restricted without a gross loss of precision. These techniques can be viewed as widenings [11] for *Pos*.

3.1 Widening scheme

As a first step, consider a *Pos* analysis in which no more than k non-ground atoms can be used to represent the formula associated with each program predicate. A simple way to carry out the analysis is as follows. Whenever a new atom a is derived for a predicate, then if there are already k atoms representing the propositional formula for that predicate, then one of these atoms a' is chosen arbitrarily and replaced by the most specific atom which subsumes both a and a'. For the special case when $k = 1$ this merely amounts taking the most specific subsumer of a new atom with the existing atom. Since we restrict our interest to positive propositional formula, we note that for $k = 1$ this implies that the single atom used to represent a formula involves only the constant *true* and variables. Moreover, we can characterize the class of formula that can be represented: these are the formula involving the symbols $\leftrightarrow$ and $\wedge$ without nesting. For example the atom $p(\_, x, true, x, \_)$ represents $p(v, w, x, y, z) \leftarrow (w \leftrightarrow y) \wedge x$. For brevity we shall denote this widened *Pos* analysis by *WPos*.

3.2 Deterministic widening scheme

As a second step, we exploit the observation that we can always safely ignore some of iff/n atoms within the body of a clause. In particular, if an *iff* predicate remains non-deterministic after solving the rest of the body, then we will ignore it. Thus we solve only deterministic calls in the clauses body. $\mathit{iff}(w, x, y, z)$ is solved deterministically as follows:

$$
\begin{array}{l}
\mathit{iff}(w, x, y, z) \leftarrow w == true, !, x = true, y = true, z = true. \\
\mathit{iff}(w, x, y, z) \leftarrow x == true, y == true, !, w = z. \\
\mathit{iff}(w, x, y, z) \leftarrow x == true, z == true, !, w = y. \\
\mathit{iff}(w, x, y, z) \leftarrow y == true, z == true, !, w = x. \\
\mathit{iff}(\_, \_, \_, \_).
\end{array}
$$

Figure 3: Deterministically solving $\mathit{iff}(w, x, y, z)$.

In practice, this approach is applied iteratively taking into account that solving an *iff* predicate may cause other *iff* predicates to become deterministic. Put another way, we perform a local fixpoint computation, terminating when no more *iff* predicates can be deterministically reduced. In some circumstances, precision can be improved by exploiting the grounding relationships within the residual *iff* predicates. In particular, a check is carried out to find *iff* predicates $iff(x, x_1, \ldots, x_n)$ and $iff(y, y_1, \ldots, y_n)$ such that $x_1 == y_1, \ldots, x_n == y_n$. Then it can be deduced that $x = y$. We call this scheme deterministic, widened *Pos* which we abbreviate it as *DPos*.

3.3 Run-time widening scheme

As a third and final step we combine the full *Pos* analysis with *DPos*, deciding at run-time whether to widen or not. The basic design of the analysis is as follows. Every time a clause needs to be evaluated, we calculate an upper bound on the degree of non-determinacy that may be required to solve the clause using classic *Pos*. If this bound is below a certain predetermined threshold, then the clause body is solved non-deterministically using *Pos*. Otherwise it is solved using the deterministic widening scheme.

The upper bound is calculated as the product of a non-determinacy level that is associated with each atom of the clause body. For user-defined atoms the non-determinacy level is defined to be the number of atoms within the database that represent that predicate. The predicates $iff/1$ and $iff/2$ are deterministic, and thus are assigned a non-determinacy level of 1. All iff/n predicates with the first argument instantiated to *true* are also given a level of 1. Otherwise an iff/n predicate is deemed to have level n.

3.4 Experimental results for goal-dependent analysis

The *Pos* analyses described above have been implemented based on the approach of [6] as outlined in section 2. The analyses are goal dependent and based on a semi-naive interpreter being applied to abstracted magic set programs. The code is written in SICStus 3.5 and the results are based on a Pentium 133MHz machine.

The table below summarizes the analysis times for *WPos* and *DPos*. The analysis times for *Pos* are also included for comparison. The *size* column gives to the number of predicates that are defined within a program. Analysis time is in seconds. The precision is measured in terms of the total number of ground models representing all the call and answer patterns. For example, the precision of $\{p^{call}(\_,\_),\ p^{ans}(true,\_)\}$ is 6 ground models. The number of ground models is a measure of the accuracy to which dependency information in inferred. The number of ground arguments is also given. For the chat program, the number of models is given as $> 10^6$ because the program that computed the number of ground models did not terminate within an hour. Similarly, analysis times for chat and trs are just given as > 3600 seconds.

program	*size*	*time*			*ground models*			*ground arguments*		
		Pos	*WPos*	*DPos*	*Pos*	*WPos*	*DPos*	*Pos*	*WPos*	*DPos*
asm	42	5.22	0.85	0.73	500	518	518	52	51	51
boyer	25	0.86	0.31	0.27	248	296	296	4	4	4
browse	16	0.37	0.14	0.13	96	108	108	42	42	42
chat	154	> 3600	7.47	7.53	–	$> 10^6$	$> 10^6$	–	504	501
cs_r	33	0.37	0.32	0.30	113	113	113	149	149	149
disj_r	29	0.31	0.27	0.28	84	84	84	97	97	97
dnf	4	0.03	0.02	0.02	11	11	11	8	8	8
gabriel	20	0.43	0.20	0.19	146	167	167	41	41	41
kalah	44	0.56	0.48	0.48	150	150	150	199	199	199
life	11	0.09	0.07	0.07	88	88	88	69	69	69
masterm	25	2.03	0.34	0.28	222	231	231	18	18	18
meta	8	0.14	0.08	0.05	50	50	50	1	1	1
nandc	14	0.65	0.22	0.20	95	98	98	15	15	15
nbody	45	3.30	0.85	0.70	623	643	661	111	111	111
peep	19	1.21	0.37	0.30	418	454	456	16	16	16
press	51	4.06	1.55	1.44	506	589	603	53	52	52
queens	5	0.06	0.04	0.02	35	43	43	3	3	3
read	40	3.70	0.83	0.80	418	423	423	102	102	102
rev	4	0.04	0.01	0.01	23	29	29	0	0	0
ronp	15	0.85	0.19	0.09	119	143	143	10	10	10
scc1	16	0.81	0.56	0.46	62	62	62	89	89	89
sdda	28	229.79	0.50	0.36	519	579	579	18	18	18
semi	21	6.92	3.91	0.68	191	203	209	30	30	27
serialize	5	0.16	0.06	0.04	58	71	71	3	3	3
treeorder	6	0.23	0.05	0.04	206	273	272	3	2	2
trs	47	> 3600	> 3600	1.39	–	–	1191	–	–	13
zebra	6	0.05	0.04	0.03	151	151	151	19	19	19

The experimental results are surprising and indicate that for a wide range of programs there is virtually no loss of groundness information when a *Pos* analysis is substituted with a *WPos* analysis. It is also surprising that for some of the programs there is not that much of a speed-up. This is due to the fact that the *iff* predicates remain non-deterministic. *DPos*, on the other hand, gives a fast and scalable analysis and obtains a significant speed-up some of the larger programs. *DPos* performs nearly as well as *WPos* for precision because the widening typically looses the information that is introduced by solving *iff* predicates non-deterministically.

The next table presents the analysis results for the run-time widening scheme for thresholds of 10, 100 and 500. These analyses are denoted Pos_{10}, Pos_{100} and Pos_{500} respectively. As before, the precision is measured in terms of number ground models representing the calls and answers, and the number of ground arguments. Although the run-time scheme is more accurate than *DPos* for a few programs, the increase in precision does not come without cost. We conclude, therefore, that *DPos* is a good compromise between precision and efficiency.

program	*time*			*ground models*			*ground arguments*		
	Pos_{10}	Pos_{100}	Pos_{500}	Pos_{10}	Pos_{100}	Pos_{500}	Pos_{10}	Pos_{100}	Pos_{500}
asm	1.54	2.27	4.27	515	508	500	???	???	52
boyer	0.64	0.83	0.95	276	249	248	???	???	4
browse	0.42	0.44	0.43	108	98	96	???	???	42
chat	14.90	36.84	49.51	$> 10^6$	$> 10^6$	$> 10^6$	501	504	504
cs_r	0.29	0.33	0.34	113	113	113	149	149	149
disj_r	0.28	0.32	0.32	84	84	84	97	97	97
dnf	0.03	0.02	0.02	11	11	11	8	8	8
gabriel	0.50	0.51	0.50	164	148	146	41	41	41
kalah	0.53	0.55	0.58	150	150	150	199	199	199
life	0.10	0.09	0.11	88	88	88	69	69	69
masterm	0.68	1.36	2.17	230	222	222	17	18	18
meta	0.11	0.15	0.17	50	50	50	1	1	1
nandc	0.51	0.56	0.65	98	96	95	15	15	15
nbody	1.44	1.82	3.09	675	667	641	104	104	108
peep	0.97	1.74	1.97	456	442	436	16	16	16
press	2.97	4.27	3.55	565	521	506	53	53	52
queens	0.07	0.07	0.08	45	39	35	2	3	3
read	1.74	2.13	2.58	422	422	418	102	102	102
rev	0.04	0.05	0.05	29	23	23	0	0	0
ronp	0.23	0.41	0.77	143	139	137	10	10	10
scc1	0.49	0.50	0.70	62	62	62	89	89	89
sdda	0.89	1.73	4.26	561	551	547	18	18	18
semi	1.27	1.47	2.48	209	209	199	27	27	27
serialize	0.10	0.14	0.22	71	67	67	3	3	3
treeorder	0.14	0.26	0.30	270	269	214	2	2	3
trs	5.63	11.71	12.17	1190	1190	900	13	13	17
zebra	0.05	0.06	0.05	151	151	151	19	19	19

(The ??? entries denote experiments that were accidently forgotten! Of course, this will be rectified if the paper is accepted for publication.)

4 Scalable definiteness analysis through *Def*

The widened *Pos* analyses perform very well on the goal-dependent analysis of logic programs. The precision of the analysis, however, can break down for applications that involve large numbers of dependencies, for example, goal-independent Prolog analysis and the analysis of constraint logic programs. *Def* is an attractive domain for reasoning about dependencies because although it cannot express disjunctive information, can always represent formulae of the form $\wedge_{i=1}^{n}(x_i \leftarrow \wedge X_i))$ where $x_i \in X$ and $X_i \subseteq X$ [1]. One problem, however, it how to implement it simply.

In this section we investigate how *Def* can be implemented straightforwardly with just linear constraints. This gives a way of implementing *Def* for even CLP($\mathcal{R}$). The key observation is that a formula like $x \wedge (y \leftarrow z)$ can be represented by the polyhedra $(x = 0) \wedge (y \leq z)$ where $0 \leq y$ and $0 \leq z$. A polyhedra is an n-dimensional space that can be expressed as the intersection of a finite set of closed half-planes. The solid square $\{\langle x, y\rangle \mid -1 \leq x \leq 1 \wedge -1 \leq y \leq 1\}$, for example, is a polyhedra since it can be represented by the half-planes $-1 \leq x$, $x \leq 1$, $-1 \leq y$ and $y \leq 1$. These half-planes are defined by non-strict inequalities and are therefore closed. The solid circle $\{\langle x, y\rangle \mid x^2 + y^2 \leq 1\}$, on the other hand, is not a polyhedra because it cannot be finitely expressed as half-planes.

The connection between definite Boolean functions and polyhedra is established through a mapping $\varphi_X : Def_X \rightarrow Poly_X$. The domain and the codomain of the mapping are parameterised by a finite, ordered variable set X. The idea is that a polyhedra, P, in n-dimensional space can be represented by a finite conjunction of (non-strict) linear inequalities, E, over a variable set X where $n = |X|$ [19]. For example, the square, $P = \{\langle x, y\rangle \mid -1 \leq x \leq 1 \wedge -1 \leq y \leq 1\}$ can be represented by $E = -1 \leq x \wedge x \leq 1 \wedge -1 \leq y \wedge y \leq 1$. This is similar to the way in which a definite Boolean function over n Boolean variables can be represented by a formulae over X such that $n = |X|$. The mapping φ_X itself is defined by:

$$\varphi_X(\wedge_{i=1}^n(x_i \leftarrow \wedge X_i)) = (\wedge_{x \in X} 0 \leq x) \wedge (\wedge_{i=1}^n x_i \leq \Sigma X_i)$$

If we suppose that $X = \{w, x, y, z\}$, then the mapping φ_X, for example, encodes the formulae $f_1 = (w \leftarrow x) \wedge (x \leftarrow y)$, $f_2 = w \leftarrow z$ and $f_3 = x \wedge (z \leftarrow (w \wedge y))$ as the polyhedra $E_1 = (0 \leq w) \wedge (0 \leq x) \wedge (0 \leq z) \wedge (w \leq x) \wedge (x \leq y)$, $E_2 = (0 \leq w) \wedge (0 \leq x) \wedge (0 \leq y) \wedge (w \leq z)$ and $E_3 = (0 \leq w) \wedge (x = 0) \wedge (0 \leq y) \wedge (0 \leq z) \wedge (z \leq w + y)$. E_3 illustrates that a grounded variable, like x, is encoded by the equation $x = 0$.

Observe that if $f \in Def_X$ then $\varphi_X(f)$ is always confined to the sub-space $\wedge_{x \in X} 0 \leq x$ called the non-negative orthant of $\mathbb{R}^n$ [19]. Furthermore, each $\varphi_X(f)$ is a cone. A cone is a space S that is closed under positive scalar multiplication. Put another way, if we write λS for $\{\lambda \vec{x} \mid \vec{x} \in S\}$ [19] then S is cone if and only if $\lambda S \subseteq S$ for all $0 < \lambda$. A space, like $\varphi_X(f)$, that is both a polyhedra and a cone, is called polyhedral cone. Finally note that each polyhedral cone $\varphi_X(f)$ includes the origin.

4.1 Conjunction operation

The encoding φ_X is such that the conjunction of linear inequations $\varphi_X(f) \wedge \varphi_X(f')$ models the groundness propagation behaviour that is expressed by the conjunction of Boolean functions $f \wedge f'$. Consider, how $E_1 \wedge E_2$, $E_1 \wedge E_3$ and $E_2 \wedge E_3$ reflect $f_1 \wedge f_2$, $f_1 \wedge f_3$ and $f_2 \wedge f_3$.

$$\begin{aligned}
E_1 \wedge E_2 &= (0 \leq w) \wedge (w \leq x) \wedge (x \leq y) \wedge \quad (w \leq z) \\
E_1 \wedge E_3 &= (w = 0) \wedge (x = 0) \wedge (0 \leq z) \wedge \quad (z \leq y) \\
E_2 \wedge E_3 &= (x = 0) \wedge (0 \leq w) \wedge (w \leq z) \wedge (z \leq w + y)
\end{aligned}$$

$$\begin{aligned}
f_1 \wedge f_2 &= (w \leftarrow x) \wedge (x \leftarrow y) \wedge (w \leftarrow z) \\
f_1 \wedge f_3 &= w \wedge x \wedge (z \leftarrow y) \\
f_2 \wedge f_3 &= x \wedge (w \leftarrow z) \wedge (z \leftarrow w \wedge y)
\end{aligned}$$

Thus the meet operator on linear inequalities can be used to simulate the meet operator in Def_X.

4.2 Convex hull operation

Polyhedral cones that include the origin possess simple computational properties can be exploited to straightforwardly simulate the join operator in Def with operations on linear inequalities. The join operator in Def does not correspond to classical disjunction but rather an upper approximation of disjunction [1]. For example, although $x \vee y$ is expressible in $Pos_{\{x,y\}}$, in $Def_{\{x,y\}}$ the most precise element that is entailed by both x and y is *true*. We follow [1] and denote the join of Def_X by $\dot{\vee}$.

Simulating $\dot{\vee}$ reduces to computing a convex hull on cones. Calculating the convex hull of two arbitrary polyhedra is surprisingly subtle. The convex hull of the point $P_1 = \{\langle 0,0\rangle\}$ and the line $P_2 = \{\langle x,y\rangle | (0 \leq x) \wedge (y = 1)\}$ is the convex space $P = \{\langle 0,1\rangle\} \cup \{\langle x,y\rangle | (0 \leq x) \wedge (0 \leq y < 1)\}$ that is *not* a polyhedra. It is not a polyhedra because the space cannot be expressed as an intersection of closed half-planes, or equivalently, as a conjunction of non-strict linear inequations.

The convex hull of two polyhedral cones P_1 and P_2 that include the origin, on the other hand, can be expressed just as $P_1 + P_2$ [19] where $P_1 + P_2 = \{\vec{x_1} + \vec{x_2} \mid \vec{x_1} \in P_1 \wedge \vec{x_2} \in P_2\}$. Moreover, $P_1 + P_2$ is also a polyhedral cone that includes the origin [5]. Computationally, if the equation sets E_1 and E_2 represent P_1 and P_2 respectively, then $P_1 + P_2 = \overline{\exists} X.E$ where $E = \rho_1(E_1) \wedge \rho_2(E_2) \wedge (\wedge_{x \in X} x = \rho_1(x) + \rho_2(x))$ and ρ_1 and ρ_2 are renamings such that the variable sets X, $\rho_1(X)$ and $\rho_2(X)$ are mutually disjoint. This gives a simple way of implementing the convex hull in terms of operations like projection $\overline{\exists}$ and renaming ρ_i that are provided in SICStus 3.5, CLP($\mathcal{R}$), ECLiPSe, *etc.*

4.3 Establishing an isomorphism

Our objective is to mimic $\wedge$ and $\dot{\vee}$ in Def_X with equivalent operations on cones. One technical problem, however, is that if $f_1, f_2 \in Def_X$ then it does not necessarily follow that there exists $f \in Def_X$ such that $\varphi_X(f) = \varphi_X(f_1) + \varphi_X(f_2)$. Consider, for example, $f_1 = (x \leftarrow u) \wedge (y \leftarrow v)$ and $f_2 = (x \leftarrow v) \wedge (y \leftarrow u)$ where $X = \{u, v, x, y\}$. Hence $\varphi_X(f_1) = (0 \leq x) \wedge (0 \leq y) \wedge (x \leq u) \wedge (y \leq v)$ and $\varphi_X(f_2) = (0 \leq x) \wedge (0 \leq y) \wedge (x \leq v) \wedge (y \leq u)$ so that $\varphi_X(f_1) + \varphi_X(f_2) = (0 \leq u) \wedge (0 \leq v) \wedge (0 \leq x) \wedge (0 \leq y) \wedge (x + y \leq u + v)$. Note, however, that $\varphi_X(f_1) + \varphi_X(f_2)$ expresses the same dependency information as $E = (0 \leq u) \wedge (0 \leq v) \wedge (0 \leq x) \wedge (0 \leq y) \wedge (x \leq u + v) \wedge (y \leq u + v)$ but E is within the image set $\varphi_X(Def_X)$. We build on this observation and introduce an equivalence relation, $\approx$, to factor together cones which express the same dependency information. The starting point is a mapping $\alpha_X : Poly_X \rightarrow Def_X$ which is defined by:

$$\alpha_X(E) = \wedge\{y' \leftarrow Y \mid E \wedge (\wedge_{y \in Y} y = 0) \models (y' = 0)\}$$

The mapping α_X is not injective and in particular $\alpha_X((0 \leq v) \wedge (0 \leq x) \wedge (0 \leq y) \wedge (x \leq u + v) \wedge (y \leq u + v)) = (x \leftarrow u \wedge v) \wedge (y \leftarrow u \wedge v) = \alpha_X((0 \leq v) \wedge (0 \leq x) \wedge (0 \leq y) \wedge (x + y \leq u + v))$. Taking this further, α_X defines a partial order $\preceq$ on $Poly_X$ by: $P_1 \preceq P_2$ if and only if $\alpha_X(P_1) \models \alpha_X(P_2)$. This, in turn, induces the equivalence relation $\approx$ on $Poly_X$ by: $P_1 \approx P_2$ iff $P_1 \preceq P_2$ and $P_2 \preceq P_1$.

With $\approx$ in place, we can define a domain $Cone_X = \{[\varphi_X(f)]_\approx \mid f \in Def_X\}$ which acts as an analogue of Def_X for cones. Since Def_X is finite, it follows that $Cone_X$ too is finite. Note, however, that an element of $Cone_X$ is itself an infinite set of polyhedra. It is possible, however, to detect the fixpoint $[E_i]_\approx = [E_{i+1}]_\approx$ without computing $\alpha_X(E_i) = f_i$ and $\alpha_X(E_{i+1}) = f_{i+1}$ and then checking that $f_i \models f_{i+1}$ and $f_{i+1} \models f_i$. The key observation is that a normal form for inequalities can be obtained by replacing inequalities with non-unit coefficients, like $E_1 = (x \leq \frac{y}{2}) \wedge (\frac{y}{2} \leq z)$, with inequalities with unit coefficients, like $E_2 = (x \leq y) \wedge (y \leq z)$. In effect, we nominate E_2 as the representative of $[E_1]_\approx$. The normal form can be computed in a single pass over the inequations. Then, providing E_i and E_{i+1} are expressed in normal form, we can detect that $[E_i]_\approx = [E_{i+1}]_\approx$ by simply checking that $E_i \models E_{i+1}$ and $E_{i+1} \models E_i$ [5].

To complete the construction of an isomorphism between Def_X and $Cone_X$ we have to lift the ordering $\preceq$ to the equivalence classes of $Poly_X/\approx$ by: $[P_1]_\approx \preceq [P_2]_\approx$ iff $P_1 \preceq P_2$. Similarly, we lift convex hull and conjunction to equivalence classes by defining: $[E_1]_\approx \sqcup [E_2]_\approx = [E_1 + E_2]_\approx$ and $[E_1]_\approx \sqcap [E_2]_\approx = [E_1 \wedge E_2]_\approx$. The construction now complete for $\langle Cone_X, \preceq, \sqcup, \sqcap \rangle$, like $\langle Def_X, \models, \dot{\vee}, \wedge \rangle$, is a finite lattice and, in addition, we have the isomorphism properties that $[\varphi_X(f_1 \dot{\vee} f_2)]_\approx = [\varphi_X(f_1)]_\approx \sqcup [\varphi_X(f_2)]_\approx$ and $[\varphi_X(f_1 \wedge f_2)]_\approx = [\varphi_X(f_1)]_\approx \sqcap [\varphi_X(f_2)]_\approx$ [5]. Stated another way, this link permits us to implement Def_X, without any loss of precision, in terms of operations on linear constraints.

4.4 Experimental results for goal-independent analysis

The semi-naive analyser of section 3 has been adapted to manipulate cones and thus compute Def_X approximations. Interestingly, the analyser is similar to the simple argument size analyser described in [4] except for the following main differences: widening is not required; the convex hull calculation is simpler; and normalisation follows head projection so that the fixpoint can be detected. We demonstrate the analysis on the Prolog benchmarks of section 3 (rather than constraint programs) because: we are interested in the scaling behaviour of the analysis and there is a dearth of large, standard benchmarks that involve constraints; and Def_X has already been shown to be a useful domain for reasoning about constraint programs [15]. We calculate goal independent Def_X approximations because this is an analysis problem for which Pos_X performs very well for accuracy and thus this acts as a strength test for Def_X. The analysis times for a one processor 55MHz SPARCstation 20 are given below complete with the number of ground models and ground arguments in the computed interpretations. As before, the precision is measured in terms of number ground instances of the atoms representing the calls and answers, and the number of ground arguments.

program	*time*		*ground models*		*ground arguments*	
	Def	*Pos*	*Def*	*Pos*	*Def*	*Pos*
ann	37.2	80.3	461	424	13	16
asm	13.8	2.8	323	304	16	16
boyer	9.7	0.4	101	87	3	6
browse	2.7	0.2	94	92	10	10
chat	116.9	1247.1	14184	6122	117	138
cs_r	38.5	13.7	316	167	23	29
disj_r	14.0	2.1	100	75	20	27
dnf	0.4	0.1	7	7	7	7
gabriel	3.5	0.2	113	110	12	14
kalah	4.9	0.5	510	263	42	51
life	5.9	0.3	164	127	15	15
masterm	1.4	0.2	57	55	19	20
meta	1.6	0.1	27	24	3	4
nandc	3.3	0.2	39	38	11	11
nbody	8.1	1.3	385	343	56	56
neural	9.4	0.4	168	161	31	31
peep	17.6	1.1	168	151	8	8
press	19.9	4.1	218	209	32	32
qplan	19.8	1.1	250	203	27	37
queens	1.5	0.1	17	17	1	1
read	13.3	3.6	339	223	27	34
reducer	50.4	11.0	732	673	23	25
rev	0.7	0.0	10	10	1	1
ronp	16.1	0.2	55	51	5	5
sdda	6.3	> 3600	299	281	4	5
semi	43.9	0.8	84	75	21	21
serialize	3.8	0.1	29	28	1	1
simple_analyzer	37.3	83.9	1310	1028	29	30
tictactoe	5.3	0.3	337	188	8	9
treeorder	13.2	0.2	56	56	0	0
trs	185.9	> 3600	630	–	7	–
zebra	1.3	0.1	14	14	12	12

The cones-based Def_X analyser is sometimes an order of magnitude slower than the model-based Pos_X analyser for the smaller benchmarks. Using cones, however, improves the scaling behaviour of the analysis since it does not suffer from the non-determinism and model growth problems that are associated with unwidened Pos_X. Consequently it outperforms Pos_X on some of the larger programs and, unlike Pos_X, always terminates. Cones will be more competitive still for systems, like CLP($\mathcal{R}$) and IF/Prolog, with fast implementations of linear constraints. In terms of precision, the model count for Def_X is often within 20% of the model count for Pos_X. This degree of precision was a surprise to us.

5 Related work

[1, 2, 9] report C-based implementations of Boolean functions. This is not the focus of our study. [10], on the other hand, report an analyser based on finite domains.

[15] considers goal-dependent analysis of CLP programs, explaining how the domain *Def* can be represented in terms of a domain $\wp(X) \times \wp(X \times \wp(\wp(X)))$

where X is the local variable set. A Herbrand constraint like $x = f(y, z)$ for example is represented by $\langle \emptyset, \{\langle x, \{\{y, z\}\}\rangle, \langle y, \{\{x\}\}\rangle, \langle z, \{\{x\}\}\rangle\}\rangle$ which encodes $x \leftrightarrow (y \wedge z)$. Abstract conjunction is expressed in terms of six rewrite rules that put conjunctions of formulae into a normal form. Although not stated, the normal form is essentially the (orthogonal) reduced monotonic body form of [1] in which a definite function is represented as $f = \wedge_{x \in X}(x \leftarrow M_x)$ where M_x is monotonic and $x \notin M_x$. Orthogonality ensures that the join ($\dot{\vee}$) is safe. Adapting an example in [1], if $f = (w \leftarrow x) \wedge (x \leftarrow y)$ and $f' = w \leftarrow z$, then the naive join is $w \leftarrow (x \wedge z)$ which is incorrect. Rule six of the rewrite system instead, however, will express f as $(w \leftarrow (x \vee y)) \wedge (x \leftarrow y)$ which leads to the correct join $w \leftarrow ((x \wedge z) \vee (y \wedge z))$. Our work shows how for goal dependent analysis of Herbrand a much simpler domain can be used. Furthermore, our cones work shows how even for language just equipped with linear constraints, we can still reap the advantages of the generalised semantics [16] and build an analyser for groundness dependencies that is simple to implement. The programmer does not, for example, need to code the rewrite rules of [15]. To be fair, however, the rewrite rule approach is likely to be more efficient.

[1] compares various normal forms of Boolean function and the relative precision of *Pos* and *Def* for a framework that calculates call patterns through a combined bottom-up and top-down analysis. The conclusion is that "positive functions generally achieve significantly higher precision than using definite functions, at small extra cost" [1]. This is indeed true for the framework of [1] (because *Def* is not condensing like *Pos*) but our work shows that for a conventional top-down framework, like GAIA [17] say, the full expressiveness of even *Def* is simply not required.

6 Future work

In the short term, our future work will focus on benchmarking our analyses on constraint programs and comparing the speed of the cones based analyser to an analyser based on Boolean constraints. In the medium term, we will investigate representations for groundness dependencies using constraints that combine the advantages of models and cones, that is, an analysis should ideally based on a deterministic constraint solver for scalability and support a fast model based approach to analysis for speed. We will also investigate more accurate ways of deciding how non-deterministic is the clause body at a given phase of the evaluation. In the long term, we will also study adapting the widening techniques to the domains and deeper semantic issues such as the interaction between numeric descriptions, like polyhedra [4] and *Sign* abstractions [7] and groundness dependencies. We also intend to work on how groundness information can be expressed using the term size norm using and thus how dependency information can be inferred through argument size analysis [4].

7 Conclusions

We have presented a widening for *Pos* and shown how the widening leads to a fast and scalable goal dependent analysis for logic programs. The analysis can be implemented easily in Prolog. We have also established an isomorphism between *Def* and cones and shown how this can be exploited to give an analysis that can be implemented straightforwardly with linear constraints.

Acknowledgments

We would like to thank Pat Hill for useful discussions and Cohavit Taboch for help with the implementation. This work was partly funded by the ESPRC Grant GR/K97635 and a Brian Spratt bursary. Much of the work was carried while Andy Heaton was visiting Ben-Gurion University.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean function for Dependency analysis. In *SAS'94*, pages 266–280. Springer-Verlag, 1994. Long version to appear in Science of Computer Programming.

[2] R. Bagnara. A Reactive Implementation of *Pos* using RORBBs. In *PLILP'96*, pages 107–121. Springer-Verlag, 1996.

[3] N. Baker and H. Søndergaard. Definiteness analysis of CLP($\mathcal{R}$). In *Australian Computer Science Conference*, pages 321–332, 1993.

[4] F. Benoy and A. King. Inferring Argument Size Relationships with CLP($\mathcal{R}$). In *LOPSTR'96*. Springer-Verlag, 1996.

[5] F. Benoy and A. King. An Isomorphism between Polyhedra and Definite Boolean Functions. Technical report, Computing Laboratory, University of Kent at Canterbury, 1997.

[6] M. Codish and B. Demoen. Analysing Logic Programs using "prop"-ositional Logic Programs and a Magic Wand. *J. of Logic Programming*, 25(3):249–274, 1995.

[7] M. Codish, A. Mulkers, M. Bruynooghe, M. García de al Banda, and M. Hermegildo. Improving Abstract Interpretations by Combining Domains. In *PEPM'93*. ACM Press, 1993.

[8] A. Cortesi, G. Filé, and W. Winsborough. Optimal Groundness Analysis using Propositional Logic. *J. of Logic Programming*, 27(1, 2 and 3):137–169, 1996.

[9] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Evaluation of the domain *Prop*. *J. of Logic Programming*, 23(3):237–278, 1995.

[10] M.-M. Cortesi, K. Musumbu, A. Rauzy, and B. Le Charlier. Efficient Bottom-up Abstract Interpretation of Prolog by means of Constraint Solving over Finite Domains. In *PLILP'93*, pages 75–91. Springer-Verlag, 1993.

[11] P. Cousot and R. Cousot. Comparing the Galois connection and Widening/Narrowing approaches to Abstract Interpretation. In *PLILP'92*, pages 269–295. Springer-Verlag, 1992.

[12] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *POPL'78*, pages 84–97, 1978.

[13] P. Dart. *Dependency Analysis and Query Interfaces to Deductive Databases*. PhD thesis, Department of Computer Science, 1988.

[14] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *PLDI'96*, pages 117–126. ACM Press, 1996.

[15] M. Garcia de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *TOPLAS*, 18(5):564–614, 1996.

[16] R. Giacobazzi, S. K. Debray, and G. Levi. A Generalised Semantics for Constraint Logic Programs. In *FGCS'92*, pages 581–591, 1992.

[17] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *TOPLAS*, 1994.

[18] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis. *ACM Lett. Program. Lang. Syst.*, 13(2–3):181–196, 1993.

[19] R. Rockafellar. *Convex Analysis.* Princeton University Press, New Jersey, 1970.