Widening Positive Boolean functions for Goal-dependent Groundness Analysis

Michael Codish[1], Andy Heaton[2], Andy King[3], Muhamed Abo-Zaed[1], Pat Hill[2]

[3]Department of Mathematics and Computer Science,
Ben-Gurion University, PoB 653, Beer-Sheba, Israel,
{abozaed,mcodish}@cs.bgu.ac.il.
[1]School of Computer Studies, University of Leeds, LS2 9JT, UK,
{heaton, hill}@scs.leeds.ac.uk.
[2]Computing Laboratory, University of Kent at Canterbury, CT2 7NF, UK,
amk@ukc.ac.uk.

Abstract. The domain of positive Boolean functions, *Pos*, is by now well established for the analysis of the variable dependencies that arise within logic programs. Analyses based on *Pos* that use binary decision diagrams have been shown to be efficient for a wide range of practical programs. However, independent of the representation, assuming that $P \neq NP$, an (unwidened) *Pos* analysis can never come with any efficiency guarantees because of its potentially explosive behaviour. This paper proposes a simple widening for (goal-dependent) groundness analysis of logic programs that guarantees scalability and efficiency. Experimental results indicate that the widening induces only a very small loss of precision.
Keywords. Logic programming, abstract interpretation, positive Boolean functions, widening.

1 Introduction

Many analyses for logic programs use Boolean functions to express dependencies between program variables. The most commonly used are the class of positive propositional formulae and its subclass of definite formulae, denoted **Pos** and **Def** respectively [1]. Numerous independent implementations have indicated that program analyses based on **Pos** and **Def** formulae are accurate and well suited to the analysis of logic programs, concurrent logic programs, constraint logic programs and deductive databases [1, 2, 5, 8, 9, 12, 13, 16, 21]. One of the best known applications of this type of analysis involves reasoning about groundness dependencies. For example, in this context the definite formula $\mathbf{x} \wedge (\mathbf{y} \leftarrow \mathbf{z})$ is interpreted to describe a program state in which $\mathbf{x}$ is definitely ground and there exists a grounding dependency such that whenever $\mathbf{z}$ becomes ground then so does $\mathbf{y}$.

A variety of techniques have been devised to represent, maintain and manipulate various classes of propositional formulae for program analyses. Worth mentioning are: (1) the use of binary decision diagrams (BDD's) and their variants, like reduced order binary decision diagrams (ROBDD's) [2, 5, 9], which lead to fast **Pos** analyses for many benchmarks; (2) the use of dual Blake canonical form (DBCF) for **Def** [1]; and (3) the use of a set of possibly non-ground atoms over the alphabet $\{\mathbf{true}, \mathbf{false}\}$ to represent the truth table of a formula [8].

The domain operations to support analyses with the first two techniques are usually implemented in C, while analyses with the third approach are easily implemented in Prolog. While the first two techniques are rather sophisticated and give reasonable performance for most programs, the only sophistication in the third technique is the simplicity in which it can be implemented. None of these iterative techniques, however, come with any efficiency guarantees. This is not surprising if we consider the number of iterations that can arise in analysis and the time/space required for each iteration. For **Pos** and **Def**, the number of iterations is exponential in the arities of predicates. The operations encountered during an iteration of the analysis based on propositional formula are typically: computing a join, computing a meet, projecting the formula (applying existential quantification), and renaming the variables in the formula. In addition, each iteration of the analysis involves a test for equivalence to determine if a fixed point has been reached. For **Pos** and for **Def** the equivalence problem is in co-NP [1, Section 4]. Put another way, in the worst-case, either the representation is of exponential size or the test for equivalence takes exponential time.

Whilst empirical evaluation results for **Pos** and **Def** presented in [1, 2, 8, 21] suggest that analyses based on **Pos** and **Def** are efficient analyses for many programs, there is no hard evidence that these analysers will be robust and efficient for all programs. In particular, it is not clear, for example, that these techniques will perform well on machine-generated predicates of very high arity like those that arise from extended definite clause grammars [15]. In more pragmatic terms, the problem is that a cautious compiler vendor is unlikely to adopt an analysis unless it comes with good performance and scalability guarantees.

In this paper we study the problem of widening **Pos**. The challenge in widening is to tune the precision of an analysis to a particular analysis application. Since most practical compiler applications require call pattern information we focus primarily on goal-dependent analysis for logic programs. As in [8] we use a set of possibly non-ground atoms over the alphabet $\{\mathbf{true}, \mathbf{false}\}$ to represent the truth table of a formula. Since the size of the truth-table can grow exponentially, widening is applied to restrain its growth.

Our surprising result is that it is sufficient to adopt a very simple representation in which a single non-ground atom alone is used to represent the formula for a call or answer pattern. This representation of the formula takes linear space and guarantees that the number of iterations in an analysis is linear in the size of the program. This approach leads to a very fast analysis that scales smoothly. Although, the class of formula that arise from this widening is not very rich, the important pragmatic result is that it is expressive enough to accommodate the backwards flow of groundness information that arises in many programs. We thus believe that the class of formulae is important. Experimental results indicate no significant loss of precision for a wide class of benchmark programs.

The rest of the paper is structured as follows. Section 2 recalls groundness analysis based on **Pos**; Section 3 reviews scalability issues of **Pos**; Section 4 introduces our widened domain; Section 5 gives the experimental evaluation; Section 6 reviews the related work; Section 7 discusses some directions for future work; and finally Section 8 presents some concluding remarks and directions for further research.

2 Groundness Analysis Based on Pos

2.1 Positive Boolean functions

We follow the convention of identifying a truth assignment with the set of variables that it maps to **true**. For brevity, we introduce the map $\mathbf{model_X(f)} = \{\mathbf{M} \subseteq \mathbf{X} \mid (\wedge\mathbf{M}) \wedge (\neg \vee \mathbf{X} \setminus \mathbf{M}) \models \mathbf{f}\}$. For example, if $\mathbf{X} = \{\mathbf{x}, \mathbf{y}\}$ then $\mathbf{model_X}(\mathbf{x} \wedge \mathbf{y}) = \{\{\mathbf{x}, \mathbf{y}\}\}$ and $\mathbf{model_X}(\mathbf{x} \vee \mathbf{y}) = \{\{\mathbf{x}\}, \{\mathbf{y}\}, \{\mathbf{x}, \mathbf{y}\}\}$. The set of positive Boolean functions over $\mathbf{X}$ is denoted $\mathbf{Pos_X}$. Recall that a Boolean function $\mathbf{f}$ is positive if $\mathbf{X} \in \mathbf{model_X(f)}$. Hence $\mathbf{x} \wedge \mathbf{y}, \mathbf{x} \vee \mathbf{y} \in \mathbf{Pos_X}$ but $\neg\mathbf{x} \notin \mathbf{Pos_X}$. The complete lattice $\mathbf{Pos_X^{\perp}} = \mathbf{Pos_X} \cup \{\mathbf{false}\}$ is ordered by set inclusion on $\mathbf{model_X(f)}$ for each Boolean function $\mathbf{f}$, with **false** and **true** as the bottom and top elements respectively. A chain in a class $\mathbf{F}$ of Boolean functions is a sequence $\mathbf{f}_1, \ldots, \mathbf{f_n}$ such that $\mathbf{f_i} \in \mathbf{F}$ for every $1 \leq \mathbf{i} \leq \mathbf{n}$ and $\mathbf{model_X(f_i)} \subset \mathbf{model_X(f_{i+1})}$ for all $1 \leq \mathbf{i} < \mathbf{n}$.

2.2 Abstracting constraints with Boolean functions

The abstract domain $\mathbf{Pos_X}$ is formally defined in [10, 19] as 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 $\mathbf{x}$ expresses that the program variable $\mathbf{x}$ is (definitely) bound to a ground term. The formula $\mathbf{x} \leftrightarrow \mathbf{y}$, on the other hand, expresses the fact that $\mathbf{x}$ is bound to a ground term if and only if $\mathbf{y}$ is, so an eventual binding of $\mathbf{x}$ to a ground term will be propagated to $\mathbf{y}$ and *vice versa*. The interested reader can find additional details on analyses with $\mathbf{Pos_X}$ in [1, 2, 8, 9, 10, 13, 19, 21].

2.3 Abstracting programs with Boolean functions

Let us first recall briefly the implementation technique for groundness dependencies that is described in [8]. 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 atom `iff(x, [x1, x2])` specifies the formula $\mathbf{x} \leftrightarrow \mathbf{x}_1 \wedge \mathbf{x}_2$ with the intended interpretation that $\mathbf{x}$ is ground if and only if $\mathbf{x}_1$ and $\mathbf{x}_2$ are. Consequently, a unification of the form $\mathbf{x} = [\mathbf{y}|\mathbf{z}]$ in the concrete program is replaced by an atom `iff(x, [y, z])`.

Similarly, a unification of the form $\mathbf{x} = []$ in the concrete program can simply be replaced by `iff(x, [])` which specifies that $\mathbf{x}$ 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 of why this is so, refer to [8, 10, 19].

Concrete `rotate`	*Abstract* `rotate`	*Auxiliary predicates*

`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).`
	`rotate(Xs,Ys) :-`
	`    append(As,Bs,Xs),`
	`    append(Bs,As,Ys).`
	
	`append(Xs,Ys,Zs) :-`
	`    iff(Xs, [ ]),`
	`    iff(Ys, [Zs]).`
	`append(Xs,Ys,Zs) :-`
	`    iff(Xs, [X,Xs1]),`
	`    iff(Zs, [X,Zs1]),`
	`    append(Xs1,Ys,Zs1).`
		`iff(true, []).`
		`iff(true, [true|Xs]) :-`
		`    iff(true, Xs).`
		`iff(false, Xs) :-`
		`    member(false,Xs).`

Fig. 1. Corresponding concrete and abstract programs.

2.4 Simple interpreter for bottom-up analysis

Figure 2 illustrates a simple Prolog interpreter (and thus an analyser) for naive bottom-up evaluation of any (finitary) logic program $\mathbf{P}$. Each clause $\mathbf{h} \leftarrow \mathbf{b_1}, \ldots, \mathbf{b_n}$ in $\mathbf{P}$ is represented as a fact of the form `user_clause`$(\mathbf{h}, [\mathbf{b_1}, \ldots, \mathbf{b_n}])$. 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`$(\mathbf{h}, \mathbf{b})$ in $\mathbf{P}$ proves the body $\mathbf{b}$ using facts derived so far and calls the predicate `cond_assert/1` which asserts the head $\mathbf{h}$ if it is new. A fact is new if it is not subsumed by any of the facts derived so far. The `iff/2` predicates introduced by the abstraction are solved as builtins using the second clause of `prove/1`. The control predicate `iterate/0`, 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. Applying the interpreter of Figure 2 to the abstract rotate program from Figure 1 gives the following atoms:

```
rotate(x,x).                append(false,_,false).
append(true,true,true).     append(_,false,false).
```

which are interpreted as representing the propositional formula $\mathtt{x}_1 \leftrightarrow \mathtt{x}_2$ and $(\mathtt{x}_1 \wedge \mathtt{x}_2) \leftrightarrow \mathtt{x}_3$ for the atoms $\mathtt{rotate(x_1,x_2)}$ and $\mathtt{append(x_1,x_2,x_3)}$ respectively. This illustrates a goal independent analysis. Goal dependent analyses are supported by applying magic sets or similar techniques. The interpreter can be used as the basis for various more efficient implementations based on semi-naive evaluation, strongly connected components, and other optimisation techniques. For further details see [7, 8].

```
iterate ← operator, fail.
iterate ← retract(flag), iterate.
iterate.

cond_assert(F) ← in_database(F), !.
cond_assert(F) ← assert(F),
    cond_assert(flag).

in_database(G) ← functor(G, N, A),
    functor(B, N, A), call(B),
    subsumes(B, G), !.
```

The "control"

```
operator ←
    user_clause(Head, Body),
    prove(Body),
    cond_assert(fact(Head)).

prove([ ]).
prove([B|Bs]) ←
    builtin(B), !,
    call(B), prove(Bs).
prove([B|Bs]) ←
    fact(B), prove(Bs).
```

The "logic"

Fig. 2. A Prolog interpreter for bottom-up evaluation.

3 (Non-)Scalability of Pos-based Analyses

The scaling problems that arise in the analysis of [8] and the interpreter depicted in Figure 2 stem from the representation of formula (as sets of atoms) and the evaluation strategy that is used within the analyser which is based on solving, in all possible ways, a clause body $\mathbf{b}_1, \ldots, \mathbf{b_n}$ using the facts derived so far for each of the calls. For each such call up to $2^{\mathbf{n}}$ ground atoms may be used to represent the corresponding propositional formula for that call.

Using a non-ground representation has some advantages. For example, the single non-ground atom `p(x, x)` represents the two ground atoms `p(true,true)` and `p(false,false)` and the single atom `p(_,...,_)` represents all of the $2^{\mathbf{n}}$ ground atoms for $\mathtt{p}/\mathbf{n}$. However, the numbers of non-ground atoms required to represent arbitrary positive propositional formula still remains exponential.

While the sophisticated implementation techniques described in [1] perform much better for large programs than the simple approach of [8], unfortunately, whatever the representation, equivalence of functions in **Pos** and **Def**, in general, is co-NP complete. This means that any representation or test for equivalence will have, in the worst case, exponential size or cost, if $\mathbf{P} \neq \mathbf{NP}$. The size of a representation is perhaps more important as it impacts on other operations, namely, computing joins and meets, projection and variable renaming. In practise, given the right variable ordering, polynomial sized representations can be obtained for many Boolean functions with the ROBDD method [1]. However, in the worst case the size of the ROBDD can grow exponentially with the number of variables (in a clause). In addition to this the longest chain in the $\mathbf{Pos}_{\perp}$ lattice is exponential. This means that the number of iterates required to compute a fixpoint for an analysis based on **Pos** can be exponential regardless of the representation used. In conclusion, an analysis based on **Pos** formulae which: restricts the representation size; and limits the maximal number of iterations to compute the fixpoint; is desirable.

4 Widening Pos

In this section we consider a simple approach to restrain the size of the propositional formula encountered in the course of a program analysis. The key idea is to restrict the number of disjuncts in the non-ground disjunctive normal form representation of the formulae. Consider the case when no more than $\mathbf{k}$ non-ground atoms can be used to represent the DNF formula associated with each program predicate. A simple way to carry out the analysis is as follows. As soon as the $\mathbf{k}+1^{\mathbf{th}}$ atom is derived for a predicate, then two of these atoms are chosen (perhaps arbitrarily) and replaced by the most specific atom which subsumes them both. These approximations can be viewed as widenings for **Pos** [11].

Since we are primarily interested in reducing the complexity of **Pos** analyses in order to obtain both fast and scalable analyses, we focus on the case where only one non-ground atom is allowed to represent each **Pos** formula. In the implementation, this merely amounts to taking the most specific subsumer of a new atom with the existing atom. Since we restrict our interest to positive formulae, we note that for $\mathbf{k} = 1$ this implies that the single atom used to represent a formula involves only the constant **true** and variables. Moreover, we can characterise the class of formula that can be represented for each predicate: these are the formula involving the symbols $\leftrightarrow$ and $\wedge$ without nesting. For example the atom `p(_, x, true, x, _)` represents the atom $\mathtt{p}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5)$ paired with the formula $(\mathbf{x}_2 \leftrightarrow \mathbf{x}_4) \wedge \mathbf{x}_3$. This sub-domain of **Pos** we name **WPos**.

WPos has similarities with the domain sometimes denoted **Con** in which the only formula allowed are conjunctions of variables. A **Con** formula can be represented by a non-ground atom without aliasing between its arguments. The aliasing that can be expressed in **WPos**, in addition, enables simple dependency information to be represented. Although **WPos** is richer than **Con** it shares the important property that its longest chain has linear length. This, in turn, bounds the number of iterations that can occur during analysis.

Lemma 1. Let $\mathbf{X}$ be a set of $\mathbf{n}$ variables. The longest chain in the lattice $\mathbf{WPos}^{\perp}_{\mathbf{X}} = \mathbf{WPos}_{\mathbf{X}} \cup \{\mathbf{false}\}$ has length $\mathbf{n}+2$. ∎

Proof. Consider an tuple $\mathbf{t}$ with arity $\mathbf{n}$ that represents a formula $\mathbf{f}$ in $\mathbf{WPos_X}$. The number of models of $\mathbf{f}$ is $2^{\mathbf{d}}$ where $\mathbf{d}$ is number of distinct variables in $\mathbf{t}$. But $\mathbf{d}$ is bounded by $\mathbf{n}$ so that a chain in $\mathbf{WPos_X}$ can, at most, contain $\mathbf{n}+1$ formulae. The longest chain in $\mathbf{WPos_X^\perp}$ thus has length $\mathbf{n}+2$. ∎

Theorem 2. The number of iterations required to obtain a fixpoint in an analysis based on the **WPos** domain is linear in the size of the program. ∎

Proof. Suppose a program contains predicates $\mathbf{p}_1, \ldots, \mathbf{p_m}$ of arity $\mathbf{n}_1, \ldots, \mathbf{n_m}$. Observer that the number of iterates for an analysis deriving success patterns is bounded from above by $\sum_{\mathbf{i}=1}^{\mathbf{m}}(\mathbf{n_i}+2)$. Similarly, the number of iterates for an analysis deriving call and answer patterns is bounded by $2 \cdot \sum_{\mathbf{i}=1}^{\mathbf{m}}(\mathbf{n_i}+2)$. ∎

The test for equivalence of **WPos** formulae boils down to a simple variance check. For completeness we show that this operation has complexity $\mathbf{O}(\mathbf{n}^2)$.

Lemma 3. The test for equivalence of formulae in **WPos** is $\mathbf{O}(\mathbf{n}^2)$. ∎

Proof. Consider a test for equivalence between two **WPos** formulae represented as non-ground tuples. The test amounts to checking which arguments are definitely ground and which arguments are aliased to each other. Namely, for two tuples of arguments $\langle \mathbf{t}_1, \ldots, \mathbf{t_n} \rangle$ and $\langle \mathbf{t}_1{}', \ldots, \mathbf{t_n}{}' \rangle$ we need to check that (1) $\mathbf{t_i} = \mathbf{true}$ iff $\mathbf{t_i}{}' = \mathbf{true}$ for all $1 \leq \mathbf{i} \leq \mathbf{n}$ and (2) $\mathbf{t_i} = \mathbf{t_j}$ iff $\mathbf{t_i}{}' = \mathbf{t_j}{}'$ for all $1 \leq \mathbf{i},\mathbf{j} \leq \mathbf{n}$. Step 1 has complexity $\mathbf{O}(\mathbf{n})$ and step 2 has complexity $\mathbf{O}(\mathbf{n}^2)$. The result follows. ∎

4.1 Implementing the basic widening scheme

It is important to note that **WPos** analyses are in particular well suited to improve the performance of the evaluation strategy described in [8]. Figure 3 shows how the interpreter given in Figure 2 can be modified to **WPos** analysis. Each time a new atom **a** is derived for a predicate, the most specific subsumer of **a** and the existing atom is found.

```
cond_assert(F) :-
    in_database(F), !.
cond_assert(F) :-
    functor(F, N, A), functor(G, N, A), fact(G) ->
        term_subsumer(F, G, D), retract(fact(G)), assert(fact(D));
        assert(fact(F)), cond_assert(flag).
```

Fig. 3. The cond_assert predicate for **WPos**.

4.2 Deterministic widening scheme

Notice that although there is never more than one solution in the database at any one time for a user predicate, multiple solutions may be derived while solving the clause due to the non-determinism in each `iff/2` atom that occurs in the clause body. For example, suppose `p(x,y) :- iff(x,[x,y])` is a clause in an abstract program. Note that the atom `iff(x,[x,y])` specifies the formula $\mathbf{x} \leftrightarrow (\mathbf{x} \wedge \mathbf{y})$ which is equivalent to $\mathbf{x} \rightarrow \mathbf{y}$. The **Pos** formula $\mathbf{x} \rightarrow \mathbf{y}$ can be represented by the set of atoms {`p(true, true)`, `p(false, _)`, `p(false, false)`}. Each of these atoms may be derived by the **WPos** analyser in the process of finding a **WPos** formula to approximate $\mathbf{x} \rightarrow \mathbf{y}$. If the atoms are derived in the order `p(true,true)` first, `p(false, _)` second and `p(false, false)` third, then `p(true, true)`, and `p(_, _)` are asserted for `p/2`. If the atoms were derived in the reverse order, three atoms would been asserted.

To overcome this problem of repeatedly checking for subsumption, and asserting and retracting atoms, we make the following observation. Consider an abstract clause with a head `p(x,y)` and a body that includes `iff(x, xs)` where **y**, say, is one of the variables in the list **xs**. If solving the call to `iff(x,xs)` contributes more than one truth value to **x** or **y** then widening will subsequently replace these values by an (unbound) variable in the single atom representing the formula for the head `p(x,y)`. It follows that if we can detect that a call to **iff**/2 remains non-deterministic then we can refrain from solving it. A call like `iff(w,[x,y,z])`, for example, can be solved deterministically as follows:

```
iff(w, [x, y, z]) :-
    w == true, !, x = true, y = true, z = true.
iff(w, [x, y, z]) :-
    x == true, y == true, !, w = z.
iff(w, [x, y, z]) :-
    x == true, z == true, !, w = y.
iff(w, [x, y, z]) :-
    y == true, z == true, !, w = x.
iff(w, [x, y, z]).
```

In practice, the **iff**/2 predicates in the body of a clause are solved deterministically and iteratively taking into account that solving one such predicate may cause others to become deterministic. Put another way, we perform a local fixpoint computation, terminating when no more **iff**/2 predicates can be deterministically reduced. Of course this approach can introduce additional approximation since not all **iff**/2 predicates will always be reduced. Applying this scheme the entire evaluation of each clause becomes fully deterministic. We call a **WPos** analysis that applies this evaluation method by **DPos**.

5 Implementation and Experimental Results

Our experiments are based on an analyser which is coded in Prolog that uses induced magic-sets [7] and eager evaluation [22] to perform efficient goal-dependent bottom-up evaluation. Induced magic is an implementation technique that avoids much of the re-computation that can arise in magic clause bodies. The affect is similar to that obtained with supplementary magic [6, 20], which avoids re-computation by maintaining and reusing partial solutions to clause bodies. However, the overhead is considerably reduced as the partial solutions are maintained implicitly by the underlying Prolog implementation. The technique is implemented as a simple meta-interpreter for goal-directed bottom-up evaluation. No magic-set transformation is needed as the dependencies between calls and answers are expressed directly within the interpreter.

Eager evaluation is described in [22]. In this approach an iteration strategy which is "close to semi-naive" is obtained as follows: whenever a new atom is derived a recursive procedure is invoked to ensure that every clause that has that atom in its body is reevaluated. In contrast to semi-naive evaluation, this means that no overhead is required for distinguishing between old and new atoms. Eager evaluation can involve slightly more re-computation than semi-naive strategy but for many applications, eager evaluation is faster than semi-naive due to the simplified data-structures. Preprocessing time is also reduced because: (1) the magic sets transformation is not required, and (2) strongly connected components do not need to be computed since eager evaluation performs a depth-first traversal of the call-graph.

The first table summarises the analysis times for **WPos** and **DPos** together with **Con** and **Pos**. The **Con** and **Pos** experiments are performed mainly for comparing precision against **WPos** and **DPos** – we are not arguing that times for **Pos** are optimal, though we have invested a lot of effort fine-tuning the **Pos** technique of [8]. The *fixpoint* columns give the time, in seconds, to compute the fixpoint whereas *total* additionally factors in the pre-processing overheads of parsing and abstraction, and for **Pos**, the post-processing overhead of summarising the call (answer) patterns for each predicate as a single call (answer) atom. Only then, for **Pos**, is the groundness information made explicit. The *precision* columns give the total number of ground arguments in the call and answer patterns. Analysis times of ∞ indicate that the fixpoint computation timed-out after five minutes. The first table is ordered with respect to number of predicates that are defined within a program. The analyser has been coded in SICStus 3.5 and the experiments have been performed on a one processor 167 MHz SPARC Ultra with 256 MByte of memory running Solaris 2.5.1.

The second table gives some statistics for our benchmarks. Most of these programs are standard with the exception of aircraft that is an air traffic verification system from the University of Huddersfield, sim is a parallelisation simulator from Manchester, and ga and tsp that are genetic algorithm and travelling salesman programs from the University of Uppsala. We have not chosen the initial, abstract queries ourselves: we use queries that were either suggested in the comments, or were annotated from previous experiments, or were supplied by the programmer. The columns *pred* and *clause* list the number of predicates and clauses in the program whereas *chain* is the length of the maximum chain the $\mathbf{WPos}^{\perp}$ lattice. (The analyser, benchmark suite and analysis results are all available from http://www.cs.ukc.ac.uk/people/staff/amk.)

file	*fixpoint*				*total*				*precision*			
	Con	DPos	WPos	Pos	Con	DPos	WPos	Pos	Con	DPos	WPos	Pos
append	0.01	0.01	0.01	0.00	0.01	0.01	0.01	0.00	3	3	3	4
rev	0.00	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0	0	0	0
dnf	0.02	0.02	0.00	0.01	0.04	0.04	0.02	0.04	1	8	8	8
queens	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	2	3	3	3
zebra	0.00	0.01	0.00	0.01	0.01	0.02	0.00	0.02	5	19	19	19
treeorder	0.01	0.01	0.01	0.08	0.02	0.02	0.02	0.09	0	0	0	0
serialize	0.01	0.01	0.02	0.02	0.02	0.02	0.03	0.03	3	3	3	3
meta	0.01	0.00	0.01	0.02	0.03	0.02	0.03	0.04	0	1	1	1
life	0.00	0.00	0.01	0.00	0.03	0.03	0.03	0.01	52	58	58	58
nandc	0.01	0.01	0.02	0.01	0.03	0.04	0.04	0.04	18	34	34	38
browse	0.01	0.01	0.02	0.02	0.03	0.03	0.03	0.04	37	42	42	42
ronp	0.02	0.02	0.07	0.15	0.04	0.04	0.09	0.18	10	10	10	10
scc1	0.03	0.01	0.03	0.02	0.06	0.05	0.06	0.06	25	92	92	92
tictactoe	0.02	0.01	0.01	0.01	0.07	0.06	0.06	0.06	48	49	49	49
gabriel	0.00	0.02	0.03	0.04	0.02	0.04	0.05	0.07	37	37	37	37
peep	0.03	0.04	0.09	0.41	0.14	0.14	0.20	0.53	8	10	10	10
tsp	0.02	0.02	0.03	0.04	0.05	0.05	0.05	0.07	72	85	85	85
boyer	0.01	0.01	0.03	0.03	0.08	0.06	0.09	0.12	18	18	18	18
mastermind	0.04	0.02	0.01	0.04	0.08	0.06	0.06	0.08	11	44	44	44
disj_r	0.04	0.01	0.02	0.01	0.07	0.05	0.05	0.04	38	97	97	97
ga	0.02	0.01	0.01	0.02	0.07	0.06	0.09	0.08	102	123	123	123
sdda	0.02	0.01	0.08	0.32	0.09	0.06	0.13	0.39	19	19	19	19
trs	0.04	0.05	∞	∞	0.17	0.19	∞	∞	14	14		
cs_r	0.06	0.02	0.04	0.04	0.14	0.08	0.11	0.11	38	149	149	149
neural	0.01	0.03	0.04	0.04	0.06	0.09	0.09	0.12	86	121	121	123
reducer	0.02	0.04	0.31	13.01	0.08	0.10	0.38	13.19	38	43	43	43
read	0.04	0.05	0.08	2.58	0.12	0.13	0.19	2.71	76	79	79	79
nand	0.20	0.04	0.07	0.09	0.30	0.15	0.19	0.21	128	288	288	288
asm	0.05	0.05	0.08	0.12	0.20	0.19	0.22	0.29	55	92	92	92
qplan	0.06	0.08	0.20	12.12	0.13	0.15	0.27	12.26	53	54	57	60
kalah	0.03	0.01	0.01	0.02	0.09	0.06	0.06	0.08	96	203	203	203
semi	0.04	0.04	5.71	8.65	0.08	0.07	5.76	8.69	21	26	26	26
sim	0.07	0.09	10.00	∞	0.24	0.28	10.18	∞	120	121	121	
ann	0.05	0.03	0.06	0.11	0.15	0.12	0.15	0.22	114	160	160	160
nbody	0.03	0.03	0.11	0.18	0.10	0.10	0.18	0.28	130	130	130	130
press	0.06	0.08	0.18	1.28	0.16	0.16	0.26	1.41	45	54	54	55
simple_analyzer	0.05	0.07	0.12	∞	0.15	0.16	0.22	∞	89	89	89	
chat_parser	0.20	0.24	0.97	∞	0.43	0.47	1.20	∞	177	233	233	
aircraft	0.81	0.15	0.29	0.18	1.28	0.64	0.77	0.70	259	637	637	637

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 only a small speed-up. This is due to the fact that the **iff** predicates remain non-deterministic. The other surprising result is that **DPos** only sacrifices a small degree of precision with respect to **WPos**. By way of contrast, simplifying **Pos** further to **Con** can have a disastrous impact on precision. Furthermore, with **DPos**, it is interesting to see that the fixpoint time is frequently dominated by the preprocessing.

file	*pred*	*clause*	*chain*	*file*	*pred*	*clause*	*chain*
aircraft	231	590	1882	ann	48	178	480
append	1	2	14	asm	44	271	406
boyer	26	136	236	browse	15	30	148
chat_parser	156	514	2104	cs_r	36	105	173
disj_r	31	48	256	dnf	5	34	36
ga	32	171	388	gabriel	18	42	184
kalah	45	84	432	life	11	26	148
mastermind	26	46	196	meta	9	30	64
nand	43	139	536	nandc	12	24	104
nbody	48	99	490	neural	39	81	364
parser	9	163	70	peep	20	227	214
press	51	162	492	qplan	44	148	468
queens	5	9	46	read	42	161	416
reducer	41	119	434	rev	4	6	30
ronp	16	34	140	scc1	16	33	178
sdda	32	86	306	semi	22	51	188
serialize	7	13	64	sim	46	186	588
simple_analyzer	67	143	776	tictactoe	16	69	170
treeorder	6	12	72	trs	35	137	348
tsp	23	38	316	zebra	6	18	62

The conclusion for **DPos** (for goal-dependent analysis at least) is that: the precision is good, first of all absolutely, in terms of the number ground arguments relative to **Pos** and, second, considering the dramatic cost/precision tradeoff. We also conclude that the analysis times are very encouraging, first of all absolutely in that no analysis exceeds 0.3 s and, second, considering that the fixpoint engine is a Prolog based meta-interpreter. In short, we have achieved a fine cost/precision tradeoff for **Pos** that can be realised in Prolog without the sophistication of BDD's.

For completeness, we have applied our analyser to goal-independent program analysis. For 24 of the benchmarks **DPos** did not loose any precision relative to **Pos** with respect to the number of ground arguments. For 10 benchmarks between 1–10% of the ground arguments were lost; whereas cs_r, read and zebra respectively lost 24%, 19% and 58% of their ground arguments. The programs nand and trs did not terminate within five minute with **Pos** but terminated within 0.2 s with **DPos**. In short, on grounds of precision, the widening does not seem appropriate for goal-independent analysis.

6 Related work

In [1] various normal forms of Boolean function and the relative precision of **Pos** and **Def** are investigated. C-based implementations of a bottom-up analyser based on each representation are described. For the representations for **Pos**, it is concluded that ROBDD's give the fastest analysis. A specialised representation for **Def**, the Dual Blake Canonical Form (DBCF), is found to be the fastest overall. For large tests it is several times faster than ROBDD's, and it is concluded that this is the representation likely to scale best for large programs. The precision achieved using **Pos** was found to be significantly higher than **Def**, although it is remarked that a top-down analyser would improve the precision of **Def** since it is not condensing.

Analysis times for implementations of **Pos** based on ROBDD's are also reported in [2, 21]. CHINA [3] takes 21.13 s on trs for the Pattern(**Pos**) domain [3, 21] with a bottom-up magic analyser that captures separate call patterns for each call site [4] on a 233 MHz Pentium II PC. For the other benchmarks that we have in common, **DPos** varies between one and five times as fast as Pattern(**Pos**) though a large proportion of this slow down is likely to stem from the Pattern component and the different framework. Our machines, too, are obviously different.

[16] considers goal-dependent analysis of CLP programs, explaining how the domain **Def** can be represented in terms of a domain $\wp(\mathbf{X}) \times \wp(\mathbf{X} \times \wp(\wp(\mathbf{X})))$ where $\mathbf{X}$ is the local variable set. A Herbrand constraint like $\mathbf{x} = \mathbf{f}(\mathbf{y}, \mathbf{z})$ for example is represented by $\langle \emptyset, \{\langle \mathbf{x}, \{\{\mathbf{y}, \mathbf{z}\}\}\rangle, \langle \mathbf{y}, \{\{\mathbf{x}\}\}\rangle, \langle \mathbf{z}, \{\{\mathbf{x}\}\}\rangle\}\rangle$ which encodes $\mathbf{x} \leftrightarrow (\mathbf{y} \wedge \mathbf{z})$. 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 $\mathbf{f} = \wedge_{\mathbf{x} \in \mathbf{X}} (\mathbf{x} \leftarrow \mathbf{M_x})$ where $\mathbf{M_x}$ is monotonic and $\mathbf{x} \notin \mathbf{M_x}$.

Debray [14] points out that each iteration of a **Pos** analysis requires the solution of a co-NP problem. He also considers the complexity implications of various precision tradeoffs, for example, bounding the number of call and answer patterns.

Genaim and Codish [17] discuss the incremental refinement of semantic based program analyses. They show that a **DPos** analysis can be incremented to a full **Pos** with no loss of precision and often for less the cost of performing the **Pos** analysis from scratch. In fact for several of the larger benchmark programs, the total cost of the two phase analysis (first **DPos**, then incrementing to **Pos**) is less than performing the original one phase **Pos** analysis.

WPos has similarities with generalised constraint propagation [18]. In [18] a generic algorithm for arbitrary constraint domains is described where successive most-specific-generalisation steps are used to approximate answers, thus avoiding the need to enumerate all the answers to a propagation constraint.

7 Future work

Our future work will focus on benchmarking our analyses on constraint programs. We will also investigate how to adapt and apply the widening to more sophisticated representations of **Pos** formulae as used in [1, 2, 5, 9]. We would also like to adapt the widening to goal-independent analysis. One way forward is to use a mixed **Pos** and **DPos** strategy that switches to **DPos** when the number of call and answer patterns begin to grow large. The problem here, in essence, is to detect the most propitious moment to change the mode of evaluation.

8 Conclusions

We have presented a widened **Pos** subdomain, **WPos**, based on restricting the size of the DNF representation of **Pos** formulae. We have shown how the relative complexity of analyses based on **WPos** is significantly reduced compared to that of **Pos** and **Def**. Our experiments indicate that the **WPos** domain is a basis for fast, scalable and goal-dependent analyses for logic programs with little loss of precision compared to **Pos**. Furthermore the analysis can be implemented straightforwardly in Prolog. In fact, once the decision is made to obtain an analysis which is guaranteed not to exhibit exponential behaviour then, we believe, there is no need to consider a representation more sophisticated than that presented in this paper.

Acknowledgements

We would like to thank Florence Benoy 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. Boolean function for Dependency analysis: Algebraic Properties and Efficient Representation. In *Static Analysis Symposium*, pages 266–280. Springer-Verlag, 1994. LNCS 864, long version to appear in Science of Computer Programming.
2. R. Bagnara. A Reactive Implementation of **Pos** using ROBDDs. In *Programming Languages: Implementations, Logics and Programs*, pages 107–121. Springer-Verlag, 1996. LNCS 1140.
3. R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages.* PhD thesis, Dipartimento di Informatica, March 1997.
4. R. Bagnara. Personal communication on analysing trs with Pattern(**Pos**). April 1998.
5. N. Baker and H. Søndergaard. Definiteness analysis for CLP($\mathcal{R}$). In *Australian Computer Science Conference*, pages 321–332, 1993.

6. C. Beeri and R. Ramakrishnan. On the power of magic. *J. Logic Programming*, 10(3&4):255–300, 1991.
7. M. Codish. Efficient Goal Directed Bottom-up Evaluation of Logic Programs. Technical report, Ben-Gurion University of the Negev, 1998. ftp://ftp.cs.bgu.ac.il/pub/people/mcodish/fastmagic.ps.gz.
8. 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.
9. M.-M. Corsini, K. Musumbu, A. Rauzy, and B. Le Charlier. Efficient Bottom-up Abstract Interpretation of Prolog by means of Constraint Solving over Finite Domains. In *Programming Language Implementation and Logic Programming*, pages 75–91. Springer-Verlag, 1993. LNCS 714.
10. A. Cortesi, G. Filé, and W. Winsborough. Optimal Groundness Analysis using Propositional Logic. *J. Logic Programming*, 27(2):137–168, 1996.
11. P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Programming Language Implementation and Logic Programming*, pages 269–295. Springer-Verlag, 1992. LNCS 631.
12. P. Dart. On Derived Dependencies and Connected Databases. *J. Logic Programming*, 11(2):163–188, 1991.
13. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems — A Case Study. In *Programming Language Design and Implementation*, pages 117–126. ACM Press, 1996.
14. S. Debray. On The Complexity of Datatflow Analysis of Logic Programs. In *Automata, Languages and Programming*, pages 509–520. Springer-Verlag, 1992. LNCS 623.
15. C. Fecht. Personal communication on **Pos**, DCG's and large arity predicates. November 1997.
16. M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM TOPLAS*, 18(5):564–614, 1996.
17. S. Genaim and M. Codish. Incremental Refinement of Semantic Based Program Analysis. Technical report, Ben-Gurion University of the Negev, 1998. ftp://ftp.cs.bgu.ac.il/pub/people/mcodish/incr.ps.gz.
18. T. Le Provost and M. Wallace. Generalized constraint propogation over the clp scheme. *J. Logic Programming*, 16(3):319–359, 1993.
19. K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Lett. Program. Lang. Syst.*, 2(4):181–196, 1993.
20. R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *J. Logic Programming*, 11(3&4):189–216, 1991.
21. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain **Prop**. *J. Logic Programming*, 23(3):237–278, 1995.
22. J. Wunderwald. Memoing Evaluation by Source-to-Source Transformation. In *Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 17–32. Springer, 1995. LNCS 1048.

This article was processed using the LaTeX macro package with LLNCS style