Implementing Groundness Analysis with Definite Boolean Functions

Jacob M. Howe and Andy King

Computing Laboratory, University of Kent, CT2 7NF, UK
{j.m.howe, a.m.king}@ukc.ac.uk

Abstract. The domain of definite Boolean functions, *Def*, can be used to express the groundness of, and trace grounding dependencies between, program variables in (constraint) logic programs. In this paper, previously unexploited computational properties of *Def* are utilised to develop an efficient and succinct groundness analyser that can be coded in Prolog. In particular, entailment checking is used to prevent unnecessary least upper bound calculations. It is also demonstrated that join can be defined in terms of other operations, thereby eliminating code and removing the need for preprocessing formulae to a normal form. This saves space and time. Furthermore, the join can be adapted to straightforwardly implement the downward closure operator that arises in set sharing analyses. Experimental results indicate that the new *Def* implementation gives favourable results in comparison with BDD-based groundness analyses.

Keywords. Abstract interpretation, (constraint) logic programs, definite Boolean functions, groundness analysis.

1 Introduction

Groundness analysis is an important theme of logic programming and abstract interpretation. Groundness analyses identify those program variables bound to terms that contain no variables (ground terms). Groundness information is typically inferred by tracking dependencies among program variables. These dependencies are commonly expressed as Boolean functions. For example, the function $x \wedge (y \leftarrow z)$ describes a state in which x is definitely ground, and there exists a grounding dependency such that whenever z becomes ground then so does y.

Groundness analyses usually track dependencies using either *Pos* [3, 4, 8, 15, 21], the class of positive Boolean functions, or *Def* [1, 16, 18], the class of definite positive functions. *Pos* is more expressive than *Def*, but *Def* analysers can be faster [1] and, in practise, the loss of precision for goal-dependent groundness analysis is usually small [18]. This paper is a sequel to [18] and is an exploration of using Prolog as a medium for implementing a *Def* analyser. The rationale for this work was partly to simplify compiler integration and partly to deliver an analyser that was small and thus easy to maintain. Furthermore, it has been suggested that the Prolog user community is not large enough to warrant a compiler vendor to making a large investment in developing an analyser. Thus any analysis that can be quickly prototyped in Prolog is particularly attractive. The main drawback of this approach has traditionally been performance.

The efficiency of groundness analysis depends critically on the way dependencies are represented. C and Prolog based *Def* analysers have been constructed around two representations: (1) Armstrong *et al* [1] argue that Dual Blake Canonical Form (DBCF) is suitable for representing *Def*. This represents functions as conjunctions of definite (propositional) clauses [12] maintained in a normal (orthogonal) form that makes explicit transitive variable dependencies. For example, the function $(x \leftarrow y) \wedge (y \leftarrow z)$ is represented as $(x \leftarrow (y \vee z)) \wedge (y \leftarrow z)$. García de la Banda *et al* [16] adopt a similar representation. It simplifies join and projection at the cost of computing and representing the (extra) transitive dependencies. Introducing redundant dependencies is best avoided since program clauses can (and sometimes do) contain large numbers of variables; the speed of analysis is often related to its memory usage. (2) King *et al* show how meet, join and projection can be implemented with quadratic operations based on a *Sharing* quotient [18]. *Def* functions are essentially represented as a set of models and widening is thus required to keep the size of the representation manageable. Widening trades precision for time and space. Ideally, however, it would be better to avoid widening by, say, using a more compact representation.

This paper contributes to *Def* analysis by pointing out that *Def* has important (previously unexploited) computational properties that enable *Def* to be implemented efficiently and coded straightforwardly in Prolog. Specifically, the paper details:

- how functions can be represented succinctly with non-ground formulae.
- how to compute the join of two formulae without preprocessing the formulae into orthogonal form [1].
- how entailment checking and Prolog machinery, such as difference lists and delay declarations, can be used to obtain a *Def* analysis in which the most frequently used domain operations are very lightweight.
- that the speed of an analysis based on non-ground formulae can compare well against BDD-based *Def* and *Pos* analyses whose domain operations are coded in C [1]. In addition, even without widening, a non-ground formulae analyser can be significantly faster than a *Sharing*-based *Def* analyser [18].

Finally, a useful spin-off of our work is a result that shows how the downward closure operator that arises in BDD-based set sharing analysis [10] can be implemented straightforwardly with standard BDD operations. This saves the implementor the task of coding another BDD operation in C.

The rest of the paper is structured as follows: Section 2 details the necessary preliminaries. Section 3 explains how join can be calculated without resorting to a normal form and also details an algorithm for computing downward closure. Section 4 investigates the frequency of various *Def* operations and explains how representing functions as (non-ground) formulae enables the frequently occurring *Def* operations to be implemented particularly efficiently using, for example, entailment checking. Section 5 evaluates a non-ground *Def* analyser against two

BDD analysers. Sections 6 and 7 describe the related and future work, and section 8 concludes.

2 Preliminaries

A Boolean function is a function $f : \mathit{Bool}^n \to \mathit{Bool}$ where $n \geq 0$. A Boolean function can be represented by a propositional formula over X where $|X| = n$. The set of propositional formulae over X is denoted by Bool_X. Throughout this paper, Boolean functions and propositional formulae are used interchangeably without worrying about the distinction [1]. The convention of identifying a truth assignment with the set of variables M that it maps to true is also followed. Specifically, a map $\psi_X(M) : \wp(X) \to \mathit{Bool}_X$ is introduced defined by: $\psi_X(M) = (\wedge M) \wedge (\neg \vee X \setminus M)$. In addition, the formula $\wedge Y$ is often abbreviated as Y.

Definition 1. The (bijective) map $\mathit{model}_X : \mathit{Bool}_X \to \wp(\wp(X))$ is defined by: $\mathit{model}_X(f) = \{M \subseteq X \mid \psi_X(M) \models f\}$.

Example 1. If $X = \{x, y\}$, then the function $\{\langle \mathit{true}, \mathit{true} \rangle \mapsto \mathit{true}, \langle \mathit{true}, \mathit{false} \rangle \mapsto \mathit{false}, \langle \mathit{false}, \mathit{true} \rangle \mapsto \mathit{false}, \langle \mathit{false}, \mathit{false} \rangle \mapsto \mathit{false}\}$ can be represented by the formula $x \wedge y$. Also, $\mathit{model}_X(x \wedge y) = \{\{x, y\}\}$ and $\mathit{model}_X(x \vee y) = \{\{x\}, \{y\}, \{x, y\}\}$.

The focus of this paper is on the use of sub-classes of Bool_X in tracing groundness dependencies. These sub-classes are defined below:

Definition 2. Pos_X is the set of positive Boolean functions over X. A function f is positive iff $X \in \mathit{model}_X(f)$. Def_X is the set of positive functions over X that are definite. A function f is definite iff $M \cap M' \in \mathit{model}_X(f)$ for all $M, M' \in \mathit{model}_X(f)$.

Note that $\mathit{Def}_X \subseteq \mathit{Pos}_X$. One useful representational property of Def_X is that each $f \in \mathit{Def}_X$ can be described as a conjunction of definite (propositional) clauses, that is, $f = \wedge_{i=1}^{n}(y_i \leftarrow Y_i)$ [12].

Example 2. Suppose $X = \{x, y, z\}$ and consider the following table, which states, for some Boolean functions, whether they are in Def_X or Pos_X and also gives model_X.

f	Def_X	Pos_X	$\mathit{model}_X(f)$
false			$\emptyset$
$x \wedge y$	•	•	$\{\{x, y\}, \{x, y, z\}\}$
$x \vee y$		•	$\{\{x\}, \{y\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$
$x \leftarrow y$	•	•	$\{\emptyset, \{x\}, \{z\}, \{x, y\}, \{x, z\}, \{x, y, z\}\}$
$x \vee (y \leftarrow z)$		•	$\{\emptyset, \{x\}, \{y\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$
true	•	•	$\{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$

Note, in particular, that $x \vee y$ is not in Def_X (since its set of models is not closed under intersection) and that false is neither in Pos_X nor Def_X.

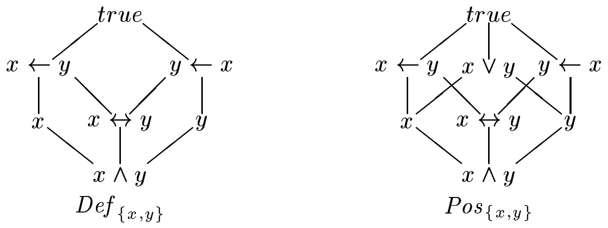

Fig. 1. Hasse diagrams

Defining $f_1 \dot\vee f_2 = \wedge\{f \in \mathit{Def}_X \mid f_1 \models f \wedge f_2 \models f\}$, the 4-tuple $\langle \mathit{Def}_X, \models, \wedge, \dot\vee \rangle$ is a finite lattice [1], where true is the top element and $\wedge X$ is the bottom element. Existential quantification is defined by Schröder's Elimination Principle, that is, $\exists x.f = f[x \mapsto \mathit{true}] \vee f[x \mapsto \mathit{false}]$. Note that if $f \in \mathit{Def}_X$ then $\exists x.f \in \mathit{Def}_X$ [1].

Example 3. If $X = \{x, y\}$ then $x \dot\vee (x \leftrightarrow y) = \wedge\{(x \leftarrow y), \mathit{true}\} = (x \leftarrow y)$, as can be seen in the Hasse diagram for dyadic Def_X (Fig. 1). Note also that $x \dot\vee y$ $= \wedge\{\mathit{true}\} = \mathit{true} \neq (x \vee y)$.

The set of (free) variables in a syntactic object o is denoted $\mathit{var}(o)$. Also, $\exists\{y_1, \ldots, y_n\}.f$ (project out) abbreviates $\exists y_1. \ldots .\exists y_n.f$ and $\overline{\exists} Y.f$ (project onto) denotes $\exists \mathit{var}(f) \setminus Y.f$. Let ρ_1, ρ_2 be fixed renamings such that $X \cap \rho_1(X) = X \cap \rho_2(X) = \rho_1(X) \cap \rho_2(X) = \emptyset$. Renamings are bijective and therefore invertible. The downward and upward closure operators $\downarrow$ and $\uparrow$ are defined by $\downarrow f = \mathit{model}_X^{-1}(\{\cap S \mid \emptyset \subset S \subseteq \mathit{model}_X(f)\})$ and $\uparrow f = \mathit{model}_X^{-1}(\{\cup S \mid \emptyset \subset S \subseteq \mathit{model}_X(f)\})$ respectively. Note that $\downarrow f$ has the useful computational property that $\downarrow f = \wedge\{f' \in \mathit{Def}_X \mid f \models f'\}$ if $f \in \mathit{Pos}_X$. Finally, for any $f \in \mathit{Bool}_X$, $\mathit{coneg}(f) = \mathit{model}_X^{-1}(\{X \setminus M \mid M \in \mathit{model}_X(f)\})$.

Example 4. Note that $\mathit{coneg}(x \vee y) = \mathit{model}_{\{x,y\}}^{-1}(\{\{x\}, \{y\}, \emptyset\})$ and therefore $\uparrow \mathit{coneg}(x \vee y) = \mathit{true}$. Hence $\mathit{coneg}(\uparrow \mathit{coneg}(x \vee y)) = \mathit{true} = \downarrow x \vee y$.

This is no coincidence as $\mathit{coneg}(\uparrow \mathit{coneg}(f)) = \downarrow f$. Therefore coneg and $\uparrow$ can be used to calculate $\downarrow$.

3 Join and downward closure

Calculating join in *Def* is not as straightforward as one would initially think, because of the problem of transitive dependencies. Suppose $f_1, f_2 \in \mathit{Def}_X$ so that $f_i = \wedge F_i$ where $F_i = \{y_1^i \leftarrow Y_1^i, \ldots, y_{n_i}^i \leftarrow Y_{n_i}^i\}$. One naive tactic to compute $f_1 \dot\vee f_2$ might be $F = \{y \leftarrow Y_j^1 \wedge Y_k^2 \mid y \leftarrow Y_j^1 \in F_1 \wedge y \leftarrow Y_k^2 \in F_2\}$. Unfortunately, in general, $\wedge F \not\models f_1 \dot\vee f_2$ as is illustrated in the following example.

Example 5. Put $F_1 = \{x \leftarrow u, u \leftarrow y\}$ and $F_2 = \{x \leftarrow v, v \leftarrow y\}$ so that $F = \{x \leftarrow u \wedge v\}$, but $f_1 \dot\vee f_2 = (x \leftarrow (u \wedge v)) \wedge (x \leftarrow y) \neq \wedge F$. Note, however, that if $F_1 = \{x \leftarrow u, u \leftarrow y, x \leftarrow y\}$ and $F_2 = \{x \leftarrow v, v \leftarrow y, x \leftarrow y\}$ then $F = \{x \leftarrow (u \wedge v), x \leftarrow (u \wedge y), x \leftarrow (v \wedge y), x \leftarrow y\}$ so that $f_1 \dot\vee f_2 = \wedge F$.

The problem is that F_i must be explicit about transitive dependencies (this idea is captured in the orthogonal form requirement of [1]). This, however, leads to redundancy in the formula which ideally should be avoided. (Formulae which not necessarily orthogonal will henceforth be referred to as non-orthogonal formulae.)

It is insightful to consider $\dot{\vee}$ as an operation on the models of f_1 and f_2. Since both $model_X(f_i)$ are closed under intersection, $\dot{\vee}$ essentially needs to extend $model_X(f_1) \cup model_X(f_2)$ with new models $M_1 \cap M_2$ where $M_i \in model_X(f_i)$ to compute $f_1 \dot{\vee} f_2$. The following definition expresses this observation and leads to a new way of computing $\dot{\vee}$ in terms of meet, renaming and projection, that does not require formulae to be first put into orthogonal form.

Definition 3. The map $\dot{\curlyvee} : {Bool_X}^2 \rightarrow Bool_X$ is defined by: $f_1 \dot{\curlyvee} f_2 = \overline{\exists} Y. f_1 \curlyvee f_2$ where $Y = var(f_1) \cup var(f_2)$ and $f_1 \curlyvee f_2 = \rho_1(f_1) \wedge \rho_2(f_2) \wedge \bigwedge_{y \in Y} y \leftrightarrow (\rho_1(y) \wedge \rho_2(y))$.

Note that $\dot{\curlyvee}$ operates on $Bool_X$ rather than Def_X. This is required for the downward closure operator. Lemma 1 expresses a key relationship between $\dot{\curlyvee}$ and the models of f_1 and f_2.

Lemma 1. Let $f_1, f_2 \in Bool_X$. $M \in model_X(f_1 \dot{\curlyvee} f_2)$ if and only if there exists $M_i \in model_X(f_i)$ such that $M = M_1 \cap M_2$.

Proof. Put $X' = X \cup \rho_1(X) \cup \rho_2(X)$.

Let $M \in model_X(f_1 \dot{\curlyvee} f_2)$. There exists $M \subseteq M' \subseteq X'$ such that $M' \in model_{X'}(f_1 \curlyvee f_2)$. Let $M_i = \rho_i^{-1}(M' \cap \rho_i(Y))$. Observe that $M \subseteq M_1 \cap M_2$ since $(\rho_1(y) \wedge \rho_2(y)) \leftarrow y$. Also observe that $M_1 \cap M_2 \subseteq M$ since $y \leftarrow (\rho_1(y) \wedge \rho_2(y))$. Thus $M_i \in model_X(f_i)$ and $M = M_1 \cap M_2$, as required.

Let $M_i \in model_X(f_i)$ and put $M = M_1 \cap M_2$ and $M' = M \cup \rho_1(M_1) \cup \rho_1(M_2)$. Observe $M' \in model_{X'}(f_1 \curlyvee f_2)$ so that $M \in model_X(f_1 \dot{\curlyvee} f_2)$. ■

From lemma 1 flows the following corollary and also the useful result that $\dot{\curlyvee}$ is monotonic.

Corollary 1. Let $f \in Pos_X$. Then $f = f \dot{\curlyvee} f$ if and only if $f \in Def_X$.

Lemma 2. $\dot{\curlyvee}$ is monotonic, that is, $f_1 \dot{\curlyvee} f_2 \models f_1' \dot{\curlyvee} f_2'$ whenever $f_i \models f_i'$.

Proof. Let $M \in model_X(f_1 \dot{\curlyvee} f_2)$. By lemma 1, there exist $M_i \in model_X(f_i)$ such that $M = M_1 \cap M_2$. Since $f_i \models f_i'$, $M_i \in model_X(f_i')$ and hence, by lemma 1, $M \in model_X(f_1' \dot{\curlyvee} f_2')$. ■

The following proposition states that $\dot{\curlyvee}$ coincides with $\dot{\vee}$ on Def_X. This gives a simple algorithm for calculating $\dot{\vee}$ that does not depend on the representation of a formula.

Proposition 1. Let $f_1, f_2 \in Def_X$. Then $f_1 \dot{\curlyvee} f_2 = f_1 \dot{\vee} f_2$.

Proof. Since $X \models f_2$ it follows by monotonicity that $f_1 = f_1 \dot{\curlyvee} X \models f_1 \dot{\curlyvee} f_2$ and similarly $f_2 \models f_1 \dot{\curlyvee} f_2$. Hence $f_1 \dot{\vee} f_2 \models f_1 \dot{\curlyvee} f_2$ by the definition of $\dot{\vee}$.

Now let $M \in model_X(f_1 \dot{\curlyvee} f_2)$. By lemma 1, there exists $M_i \in model_X(f_i)$ such that $M = M_1 \cap M_2 \in model_X(f_1 \dot{\vee} f_2)$. Hence $f_1 \dot{\curlyvee} f_2 \models f_1 \dot{\vee} f_2$. ■

Downward closure is closely related to $\dot{\curlyvee}$ and, in fact, $\dot{\curlyvee}$ can be used repeatedly to compute a finite iterative sequence that converges to $\downarrow$. This is stated in proposition 2. Finiteness follows from bounded chain length of Pos_X.

Proposition 2. Let $f \in Pos_X$. Then $\downarrow f = \vee_{i \geq 1} f_i$ where $f_i \in Pos_X$ is the increasing chain given by: $f_1 = f$ and $f_{i+1} = f_i \dot{\curlyvee} f_i$.

Proof. Let $M \in model_X(\downarrow f)$. Thus there exists $M_j \in model_X(f)$ such that $M = \cup_{j=1}^{m} M_j$. Observe $M_1 \cap M_2, M_3 \cap M_4, \ldots \in model_X(f_2)$ and therefore $M \in model_X(f_{\lceil \log_2(m) \rceil})$. Since $m \leq 2^{2^n}$ where $n = |X|$ it follows that $\downarrow f \models f_{2^n}$.

Proof by induction is used for the opposite direction. Observe that $f_1 \models \downarrow f$. Suppose $f_i \models \downarrow f$. Let $M \in model_X(f_{i+1})$. By lemma 1 there exists $M_1, M_2 \in model_X(f_i)$ such that $M = M_1 \cap M_2$. By the inductive hypothesis $M_1, M_2 \in model_X(\downarrow f)$ thus $M \in model_X(\downarrow f)$. Hence $f_{i+1} \models \downarrow f$.

Finally, $\vee_{i=1} f_i \in Def_X$ since $f_1 \in Pos_X$ and $\dot{\curlyvee}$ is monotonic and thus $X \in model_X(\vee_{i=1} f_i)$. ■

The significance of this is that it enables $\downarrow$ to be computed in terms of existing BDD operations thus freeing the implementor from more low level coding.

4 Design and implementation

There are typically many degrees of freedom in designing an analyser, even for a given domain. Furthermore, work can often be shifted from one abstract operation into another. For example, García de la Banda *et al* [16] maintain DBCF by a meet that uses six rewrite rules to normalise formulae. This gives a linear time join and projection at the expense of an exponential meet. Conversely, King *et al* [18] have meet, join and projection operations that are quadratic in the number of models. Note, however, that the numbers of models is exponential (explaining the need for widening). Ideally, an analysis should be designed so that the most frequently used operations have low complexity and are therefore fast.

4.1 Frequency analysis

In order to balance the frequency of an abstract operation against its cost, a BDD-based *Def* analyser was implemented and instrumented to count the number of calls to the various abstract operations. The BDD-based *Def* analyser is coded in Prolog as a simple meta-interpreter that uses induced magic-sets [7] and eager evaluation [22] to perform goal-dependent bottom-up evaluation.

Induced magic is a refinement of the magic set transformation, avoiding much of the re-computation that arises because of the repetition of literals in the bodies of magicked clauses [7]. It also avoids the overhead of applying the magic set transformation. Eager evaluation [22] is a fixpoint iteration strategy which proceeds as follows: whenever an atom is updated with a new (less precise) abstraction, a recursive procedure is invoked to ensure that every clause that has that atom in its body is re-evaluated. Induced magic may not be as efficient as, say, GAIA [19] but it can be coded easily in Prolog.

The BDD-based *Def* analysis is built on a ROBDD package coded by Armstrong and Schachte [1]. The package is intended for *Pos* analysis and therefore supplies a $\vee$ join rather than a $\dot{\vee}$ join. The package did contain, however, a hand-crafted C upward closure operator $\uparrow$ enabling $\dot{\vee}$ to be computed by $f_1 \dot{\vee} f_2 = \downarrow(f_1 \vee f_2)$ where $\downarrow f = coneg(\uparrow coneg(f))$. The operation $coneg(f)$ can be computed simply by interchanging the left and right (true and false) branches of an ROBDD. The analyser also uses the environment trimming tactic used by Schachte to reduce the number of variables that occur in a ROBDD. Specifically, clause variables are numbered and each program point is associated with a number, in such a way that if a variable has a number less than that associated with the program point, then it is redundant (does not occur to the right of the program point) and hence can be projected out. This optimisation is important in achieving practical analysis times for some large programs.

The following table gives a breakdown of the number of calls to each abstract operation in the BDD-based *Def* analysis of eight large programs. Meet, join, equiv, project and rename are the obvious Boolean operations. Join (diff) is the number of calls to a join $f_1 \dot{\vee} f_2$ where $f_1 \dot{\vee} f_2 \neq f_1$ and $f_1 \dot{\vee} f_2 \neq f_2$. Project (trim) are the number of calls to project that stem from environment trimming.

file	strips	chat_parser	sim_v5-2	peval	aircraft	essln	chat_80	aqua_c
meet	815	4471	2192	2198	7063	8406	15483	112455
join	236	1467	536	632	2742	1668	4663	35007
join (diff)	33	243	2	185	26	177	693	5173
equiv	236	1467	536	632	2742	1668	4663	35007
project	330	1774	788	805	3230	2035	5523	38163
project (trim)	173	1384	770	472	2082	2376	5627	42989
rename	857	4737	2052	2149	8963	5738	14540	103795

Observe that meet and rename are called most frequently and therefore, ideally, should be the most lightweight. Project, project (trim), join and equiv calls occur with similar frequency but note that it is rare for a join to differ from both its arguments. Join is always followed by an equivalence and this explains why the join and equiv rows coincide.

Next, the complexity of ROBDD and DBCF (specialised for *Def* [1]) operations are reviewed in relation to their calling frequency. Suggestions are made about balancing the complexity of an operation against its frequency by using a non-orthogonal formulae representation.

For ROBDDs (DBCF) meet is quadratic (exponential) in the size of its arguments [1]. For ROBDDs (DBCF) these arguments are exponential (polynomial) in the number of variables. Representing *Def* functions as non-orthogonal formulae is attractive since meet is concatenation which can be performed in constant time (using difference lists). Renaming is quadratic for ROBDDs (linear for DBCF) in the size of its argument [1]. Renaming a non-orthogonal formula is $O(m \log(n))$ where m (n) is the number of symbols (variables) in its argument.

For ROBDDs (DBCF), join is quadratic (quartic) in the size of its arguments [1]. For non-orthogonal formulae, join is exponential. Note, however, that the majority of joins result in one of the operands and hence are unnecessary. This can be detected by using an entailment check which is quadratic in the size of the representation. Thus it is sensible to filter join through an entailment check so that join is called comparatively rarely. Therefore its complexity is less of an issue. Specifically, if $f_1 \models f_2$ then $f_1 \dot{\vee} f_2 = f_2$. For ROBDDs, equivalence checking is constant time, whereas for DBCF it is linear in the size of the representation. For non-orthogonal formulae, equivalence is quadratic in the size of the representation. Observe that meet occurs more frequently than equality and therefore a gain should be expected from trading an exponential meet and a linear join for a constant time meet and an exponential join.

For ROBDDs (DBCF), projection is quadratic (linear) in the size of its arguments [1]. For a non-orthogonal representation, projection is exponential, but again, entailment checking can be used to prevent the majority of projections.

4.2 The GEP representation

A call (or answer) pattern is a pair $\langle a, f \rangle$ where a is an atom and $f \in Def_{var(a)}$. Normally the arguments of a are distinct variables. The formula f is a conjunction (list) of propositional Horn clauses in the *Def* analysis described in this paper. In a non-ground representation the arguments of a can be instantiated and aliased to express simple dependency information [9]. For example, if $a = p(x_1, ..., x_5)$, then the atom $p(x_1, true, x_1, x_4, true)$ represents a coupled with the formula $(x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5$. This enables the abstraction $\langle p(x_1, ..., x_5), f_1 \rangle$ to be collapsed to $\langle p(x_1, true, x_1, x_4, true), f_2 \rangle$ where $f_1 = (x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5 \wedge f_2$. This encoding leads to a more compact representation and is similar to the GER factorisation of ROBDDs proposed by Bagnara and Schachte [3]. The representation of call and answer patterns described above is called GEP (groundness, equivalences and propositional clauses) where the atom captures the first two properties and the formula the latter. Note that the current implementation of the GEP representation does not avoid inefficiencies in the representation such as the repetition of *Def* formulae.

4.3 Abstract operations

The GEP representation requires the abstract operations to be lifted from Boolean formulae to call and answer patterns.

Meet The meet of the pairs $\langle a_1, f_1 \rangle$ and $\langle a_2, f_2 \rangle$ can be computed by unifying a_1 and a_2 and concatenating f_1 and f_2.

Renaming The objects that require renaming are formulae and call (answer) pattern GEP pairs. If a dynamic database is used to store the pairs [17], then renaming is automatically applied each time a pair is looked-up in the database. Formulae can be renamed with a single call to the Prolog builtin copy_term.

Join Calculating the join of the pairs $\langle a_1, f_1\rangle$ and $\langle a_2, f_2\rangle$ is complicated by the way that join interacts with renaming. Specifically, in a non-ground representation, call (answer) patterns would be typically stored in a dynamic database so that $var(a_1) \cap var(a_2) = \emptyset$. Hence $\langle a_1, f_1\rangle$ (or equivalently $\langle a_2, f_2\rangle$) have to be appropriately renamed before the join is calculated. This is achieved as follows. Plotkin's anti-unification algorithm [20] is used to compute the most specific atom a that generalises a_1 and a_2. The basic idea is to reformulate a_1 as a pair $\langle a'_1, f'_1\rangle$ which satisfies two properties: a'_1 is a syntactic variant of a; the pair represents the same dependency information as $\langle a_1, true\rangle$. A pair $\langle a'_2, f'_2\rangle$ is likewise constructed that is a reformulation of a_2. The atoms a, a'_1 and a'_2 are unified and then the formula $f = (f_1 \wedge f'_1) \dot{\curlyvee} (f_2 \wedge f'_2)$ is calculated as described in section 3 to give the join $\langle a, f\rangle$. In actuality, the computation of $\langle a'_1, f'_1\rangle$ and the unification $a = a'_1$ can be combined in a single pass as is outlined below. Suppose $a = p(t_1, \ldots, t_n)$ and $a_1 = p(s_1, \ldots, s_n)$. Let $g_0 = true$. For each $1 \leq k \leq n$, one of the following cases is selected. (1) If t_k is syntactically equal to s_k, then put $g_k = g_{k-1}$. (2) If s_k is bound to $true$, then put $g_k = g_{k-1} \wedge (t_k \leftarrow true)$. (3) If $s_k \in var(\langle s_1, \ldots, s_{k-1}\rangle)$, then unify s_k and t_k and put $g_k = g_{k-1}$. (4) Otherwise, put $g_k = g_{k-1} \wedge (t_k \leftarrow s_k) \wedge (s_k \leftarrow t_k)$. Finally, let $f'_1 = g_n$. The algorithm is applied analogously to bind variables in a and construct f'_2. The join of the pairs is then given by $\langle a, (f_1 \wedge f'_1) \dot{\curlyvee} (f_2 \wedge f'_2)\rangle$.

Example 6. Consider the join of the GEP pairs $\langle a_1, true\rangle$ and $\langle a_2, y_1 \leftarrow y_2\rangle$ where $a_1 = p(true, x_1, x_1, x_1)$ and $a_2 = p(y_1, y_2, true, true)$. The most specific generalisation of a_1 and a_2 is $a = p(z_1, z_2, z_3, z_3)$. The table below illustrates the construction of $\langle a'_1, f'_1\rangle$ and $\langle a'_2, f'_2\rangle$ in the left- and right-hand columns.

k	$case$	g_k	θ_k	$case'$	g'_k	θ'_k
0		$true$	ϵ		$true$	ϵ
1	2	$z_1 \leftarrow true$	ϵ	4	$y_1 \leftrightarrow z_1$	ϵ
2	4	$g_1 \wedge (z_2 \leftrightarrow x_1)$	θ_1	4	$g'_1 \wedge (y_2 \leftrightarrow z_2)$	θ_1
3	3	g_2	$\{x_1 \mapsto z_3\}$	2	$g'_2 \wedge (z_3 \leftarrow true)$	θ_1
4	1	g_2	θ_3	2	$g'_3 \wedge (z_3 \leftarrow true)$	θ_1

Putting $\theta = \theta'_4 \circ \theta_4 = \{x_1 \mapsto z_3\}$, the join is given by $\langle \theta(a), \theta(g_4 \wedge true) \dot{\curlyvee} \theta(g'_4 \wedge y_1 \leftarrow y_2)\rangle = \langle a, (z_1 \leftarrow true) \wedge (z_2 \leftrightarrow z_3) \dot{\curlyvee} (y_1 \leftrightarrow z_1) \wedge (y_2 \leftrightarrow z_2) \wedge (z_3 \leftarrow true) \wedge (y_1 \leftarrow y_2)\rangle = \langle p(z_1, z_2, z_3, z_3), (z_1 \leftarrow z_2) \wedge (z_3 \leftarrow z_2)\rangle$.

Note that often a_1 is a variant of a_2. This can be detected with a lightweight variance check, enabling join and renaming to be reduced to unifying a_1 and a_2 and computing $f = f_1 \dot{\curlyvee} f_2$ to give the pair $\langle a_1, f\rangle$.

Projection Projection is only applied to formulae. Each of the variables to be projected out is eliminated in turn, as follows. Suppose x is to be projected out of f. First, all those clauses with x as their head are found, giving $\{x \leftarrow X_i \mid i \in I\}$ where I is a (possibly empty) index set. Second, all those clauses with x in the body are found, giving $\{y \leftarrow Y_j \mid j \in J\}$ where J is a (possibly empty) index set. Thirdly these clauses of f are replaced by $\{y \leftarrow Z_{i,j} \mid i \in I \wedge j \in J \wedge Z_{i,j} = X_i \cup (Y_j \setminus \{x\}) \wedge y \notin Z_{i,j}\}$ (syllogizing). Fourthly, a compact representation is maintained by eliminating redundant clauses (absorption). By appropriately ordering the clauses, all four steps can be performed in a single pass over f. A final pass over f retracts clauses such as $x \leftarrow true$ by binding x to true and also removes clause pairs such as $y \leftarrow z$ and $z \leftarrow y$ by unifying y and z.

Entailment Entailment checking is only applied to formulae. A forward chaining decision procedure for propositional Horn clauses (and hence *Def*) is used to test entailment. A non-ground representation allows chaining to be implemented efficiently using block declarations. To check that $\wedge_{i=1}^{n} y_i \leftarrow Y_i$ entails $z \leftarrow Z$ the variables of Z are first grounded. Next, a process is created for each clause $y_i \leftarrow Y_i$ that blocks until Y_i is ground. When Y_i is ground, the process resumes and grounds y_i. If z is ground after a single pass over the clauses, then $(\wedge_{i=1}^{n} y_i \leftarrow Y_i) \models z \leftarrow Z$. By calling the check under negation, no problematic bindings or suspended processes are created.

5 Experimental evaluation

A *Def* analyser using the non-ground techniques described in this paper has been implemented. This implementation is built in Prolog using the same induced magic framework as for the BDD-based *Def* analyser, therefore the analysers work in lock step and generate the same results. (The only difference is that the non-ground analyser does not implement environment trimmed since the representation is far less sensitive to the number of variables in a clause.) The core of the analyser (the fixpoint engine) is approximately 400 lines of code and took one working week to write, debug and tune.

In order to investigate whether entailment checking, the join ($\dot{\curlyvee}$) algorithm, and the GEP representation are enough to obtain a fast and scalable analysis, the non-ground analyser was compared with the BDD-based analyser for speed and scalability. Since King *et al* [18] do not give precision results for *Pos* for larger benchmarks, we have also implemented a BDD-based *Pos* analyser in the same vein, so that firmer conclusions about the relative precision of *Def* and *Pos* can be drawn. It is reported in [2], [3] that a hybrid implementation of ROBDDs, separating maintenance of definiteness information and of various forms of dependency information can give significantly improved performance. Therefore, it is to be expected that an analyser based on such an implementation of ROBDDs would be faster than that used here.

The comparisons focus on goal-dependent groundness analysis of 60 Prolog and CLP($\mathcal{R}$) programs. The results are given in the table below. In this table, the size column gives the number of distinct (abstract) clauses in the programs. The abs column gives the time for parsing the files and abstracting them, that is, replacing built-ins, such as arg(x, t, s), with formulae, such as $x \wedge (s \leftarrow t)$.

file	size	abs	fixpoint Def_{NG}	Def_{BDD}	Pos	precision Def	Pos	%
rotate.pl	3	0.00	0.00	0.00	0.00	3	6	50
circuit.clpr	20	0.02	0.02	0.03	0.02	3	3	0
air.clpr	20	0.02	0.02	0.03	0.02	9	9	0
dnf.clpr	23	0.02	0.01	0.01	0.01	8	8	0
dcg.pl	23	0.02	0.01	0.01	0.02	59	59	0
hamiltonian.pl	23	0.02	0.01	0.01	0.01	37	37	0
poly10.pl	29	0.02	0.00	0.00	0.01	0	0	0
semi.pl	31	0.03	0.03	0.28	0.28	28	28	0
life.pl	32	0.02	0.01	0.02	0.02	58	58	0
rings-on-pegs.clpr	34	0.02	0.02	0.04	0.04	11	11	0
meta.pl	35	0.01	0.01	0.02	0.01	1	1	0
browse.pl	36	0.02	0.01	0.02	0.02	41	41	0
gabriel.pl	38	0.02	0.01	0.03	0.03	37	37	0
tsp.pl	38	0.03	0.01	0.04	0.04	122	122	0
nandc.pl	40	0.03	0.01	0.03	0.03	37	37	0
csg.clpr	48	0.04	0.01	0.02	0.02	12	12	0
disj_r.pl	48	0.02	0.01	0.04	0.04	97	97	0
ga.pl	48	0.06	0.01	0.04	0.04	141	141	0
critical.clpr	49	0.03	0.03	0.04	0.04	14	14	0
scc1.pl	51	0.03	0.01	0.06	0.04	89	89	0
mastermind.pl	53	0.04	0.01	0.04	0.04	43	43	0
ime_v2-2-1.pl	53	0.04	0.03	0.09	0.08	101	101	0
robot.pl	53	0.03	0.00	0.01	0.01	41	41	0
cs_r.pl	54	0.05	0.01	0.04	0.04	149	149	0
tictactoe.pl	56	0.06	0.01	0.03	0.04	60	60	0
flatten.pl	56	0.03	0.04	0.09	0.08	27	27	0
dialog.pl	61	0.02	0.01	0.03	0.03	70	70	0
map.pl	66	0.02	0.01	0.08	0.08	17	17	0
neural.pl	67	0.05	0.01	0.05	0.05	123	123	0
bridge.clpr	69	0.08	0.01	0.02	0.03	24	24	0
conman.pl	71	0.04	0.00	0.02	0.02	6	6	0
kalah.pl	78	0.04	0.02	0.04	0.04	199	199	0
unify.pl	79	0.04	0.07	0.12	0.10	70	70	0
nbody.pl	85	0.07	0.06	0.10	0.11	113	113	0
peep.pl	86	0.11	0.03	0.06	0.05	10	10	0
boyer.pl	95	0.06	0.04	0.04	0.05	3	3	0
bryant.pl	95	0.07	0.20	0.15	0.15	99	99	0
sdda.pl	99	0.05	0.06	0.06	0.06	17	17	0
read.pl	105	0.07	0.06	0.11	0.10	99	99	0
press.pl	109	0.07	0.11	0.16	0.18	53	53	0
qplan.pl	109	0.08	0.02	0.08	0.07	216	216	0
trs.pl	111	0.11	0.11	0.31	0.60	13	13	0
reducer.pl	113	0.07	0.11	0.16	0.14	41	41	0
simple_analyzer.pl	140	0.09	0.13	0.34	0.44	89	89	0
dbqas.pl	146	0.09	0.02	0.05	0.05	43	43	0
ann.pl	148	0.09	0.11	0.24	0.23	74	74	0
asm.pl	175	0.14	0.06	0.14	0.13	90	90	0
nand.pl	181	0.12	0.04	0.21	0.19	402	402	0
rubik.pl	219	0.16	0.15	0.39	0.40	158	158	0
lnprolog.pl	221	0.10	0.08	0.14	0.14	143	143	0
ili.pl	225	0.15	0.25	0.23	0.24	4	4	0
sim.pl	249	0.18	0.39	0.56	0.52	100	100	0
strips.pl	261	0.17	0.01	0.11	0.11	142	142	0
chat_parser.pl	281	0.21	0.45	0.59	0.60	505	505	0
sim_v5-2.pl	288	0.17	0.05	0.20	0.20	455	457	0.4
peval.pl	328	0.16	0.28	0.27	0.27	27	27	0
aircraft.pl	397	0.48	0.14	0.55	0.59	687	687	0
essln.pl	565	0.36	0.21	0.58	0.58	163	163	0
chat_80.pl	888	0.92	1.31	1.89	2.27	855	855	0
aqua_c.pl	4009	2.48	11.29	104.99	897.10	1288	1288	0

The abstracter deals with meta-calls, asserts and retracts following the elegant (two program) scheme detailed by Bueno *et al* [6]. The fixpoint columns give the time, in seconds, to compute the fixpoint for each of the three analysers (Def_{NG} and Def_{BDD} denote respectively the non-ground and BDD-based *Def* analyser). The precision columns give the total number of ground arguments in the call and answer patterns (and exclude those ground arguments for predicates introduced by normalising the program into definite clauses). The % column express the loss of precision by *Def* relative to *Pos*. All three analysers were coded in SICStus 3.7 and the experiments performed on a 296MHz Sun UltraSPARC-II with 1GByte of RAM running Solaris 2.6.

The experimental results indicate the precision of *Def* is close to that of *Pos*. Although rotate.pl is small it has been included in the table because it was the only program for which significant precision was lost. Thus, whilst it is always possible to construct programs in which disjunctive dependency information (which cannot be traced in *Def*) needs to be tracked to maintain precision, these results suggest that *Def* is adequate for top-down groundness analysis of many programs.

The speed of the non-ground *Def* analyser compares favourably with both the BDD analysers. This is surprising because the BDD analysers make use of hashing and memoisation to avoid repeated work. In the non-ground *Def* analyser, the repeated work is usually in meet and entailment checking, and these operations are very lightweight. In the larger benchmarks, such as aqua_c.pl, the BDD analysis becomes slow as the BDDs involved are necessarily large. Widening for BDDs can make such examples more manageable [15]. Notice that the time spent in the core analyser (the fixpoint engine) is of the same order as that spent in the abstracter. This suggests that a large speed up in the analysis time needs to be coupled with a commensurate speedup in the abstracter.

To give an initial comparison with the *Sharing*-based *Def* analyser of King *et al* [18], the clock speed of the Sparc-20 used in the *Sharing* experiments has been used to scale the results in this paper. These findings lead to the preliminary conclusion that the analysis presented in this paper is about twice as fast as the *Sharing* quotient analyser. Furthermore, this analyser relies on widening to keep the abstractions small, hence may sacrifice some precision for speed.

6 Related work

Van Hentenryck *et al* [21] is an early work which laid a foundation for BDD-based *Pos* analysis. Corsini *et al* [11] describe how variants of *Pos* can be implemented using Toupie, a constraint language based on the μ-calculus. If this analyser was extended with, say, magic sets, it might lead to a very respectable goal-dependent analysis. More recently, Bagnara and Schachte [3] have developed the idea [2] that a hybrid implementation of a ROBDD that keeps definite information separate from dependency information is more efficient than keeping the two together. This hybrid representation can significantly decrease the size of an ROBDD and thus is a useful implementation tactic.

Armstrong *et al* [1] study a number of different representations of Boolean function for both *Def* and *Pos*. An empirical evaluation on 15 programs suggests that specialising Dual Blake Canonical Form (DBCF) for *Def* leads to the fastest analysis overall. This representation of a *Def* function f is in orthogonal form since it is constructed from *all* the prime consequents that are entailed by f. It thus includes redundant transitive dependencies. Armstrong *et al* [1] also perform interesting precision experiments. *Def* and *Pos* are compared, however, in a bottom-up framework that is based on condensing which is therefore biased towards *Pos*. The authors point out that a top-down analyser would improve the precision of *Def* relative to *Pos* and our work supports this remark.

García de la Banda *et al* [16] describe a Prolog implementation of *Def* that is also based on an orthogonal DBCF representation (though this is not explicitly stated) and show that it is viable for some medium sized benchmarks. Fecht [15] describes another groundness analyser that is not coded in C. Fecht adopts ML as a coding medium in order to build an analyser that is declarative and easy to maintain. He uses a sophisticated fixpoint solver and his analysis times compare favourably with those of Van Hentenryck *et al* [21].

Codish and Demoen [8] describe a non-ground model based implementation technique for *Pos* that would encode $x_1 \leftrightarrow (x_2 \wedge x_3)$ as three tuples $\langle true, true, true\rangle$, $\langle false, \_, false\rangle$, $\langle false, false, \_\rangle$. Codish *et al* [9] propose a sub-domain of *Def* that can only propagate dependencies of the form $(x_1 \leftrightarrow x_2) \wedge x_3$ across procedure boundaries. The main finding of Codish *et al* [9] is that this sub-domain loses only a small amount of precision for goal-dependent analysis.

King *et al* [18] show how the equivalence checking, meet and join of *Def* can be efficiently computed with a *Sharing* quotient. Widening is required to keep the representation manageable.

Finally, a curious connection exists between the join algorithm described in this paper and a relaxation that occurs in disjunctive constraint solving [14]. The relaxation computes the join (closure of the convex hull) of two polyhedra P_1 and P_2 where $P_i = \{\boldsymbol{x} \in \mathbb{R}^n \mid A_i\boldsymbol{x} \leq B_i\}$. The join of P_1 and P_2 can be expressed as:

$$P = \left\{ \boldsymbol{x} \in \mathbb{R}^n \,\middle|\, \begin{array}{c} A_1\rho_1(\boldsymbol{x}) \leq B_1 \wedge A_2\rho_2(\boldsymbol{x}) \leq B_2 \wedge \\ 0 \leq \lambda \leq 1 \wedge \boldsymbol{x} = \lambda\rho_1(\boldsymbol{x}) + (1-\lambda)\rho_2(\boldsymbol{x}) \end{array} \right\}$$

which amounts to the same tactic of constructing join in terms of meet (conjunction of linear equations), renaming (ρ_1 and ρ_2) and projection (the variables of interest are $\boldsymbol{x}$).

7 Future work

Initial profiling has suggested that a significant proportion of the analysis time is spent projecting onto (new) call and answer patterns, so recoding this operation might impact on the speed of the analysis. Also, a practical comparison with a DBCF analyser would be insightful. This is the immediate future work. In the medium term, it would be interesting to apply widening to obtain an analysis with polynomial guarantees. Time complexity relates to the maximum number of iterations of a fixpoint analysis and this, in turn, depends on the length of the longest ascending chain in the underlying domain. For both Pos_X and Def_X the longest chains have length $2^n - 1$ where $|X| = n$ [18]. One way to accelerate the analysis, would be to widen call and answer patterns by discarding the formulae component of the GEP representation if the number of updates to a particular call or answer pattern exceeded, say, 8 [18]. The abstraction then corresponds to an $EPos_X$ function whose chain length is linear in X [9]. Although widening for space is not as critical as in [18], this too would be a direction for future work. In the long term, it would be interesting to apply *Def* to other dependency analysis problems, for example, strictness [13] and finiteness [5] analysis.

The frequency analysis which has been used in this paper to tailor the costs of the abstract operations to the frequency with which they are called could be applied to other analyses, such as type, freeness or sharing analyses.

8 Conclusions

The representation and abstract operations for *Def* have been chosen by following a strategy. The strategy was to design an implementation so as to ensure that the most frequently called operations are the most lightweight. Previously unexploited computational properties of *Def* have been used to avoid expensive joins (and projections) through entailment checking; and to keep abstractions small by reformulating join in such a way as to avoid orthogonal reduced monotonic body form. The join algorithm has other applications such as computing the downward closure operator that arises in BDD-based set sharing analysis.

By combining the techniques described in this paper, an analyser has been constructed that is precise, can be implemented easily in Prolog, and whose speed compares favourably with BDD-based analysers.

Acknowledgements We thank Mike Codish, Roy Dyckhoff and Andy Heaton for useful discussions. We would also like to thank Peter Schachte for help with his BDD analyser. This work was funded partly by EPSRC Grant GR/MO8769.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. R. Bagnara. A Reactive Implementation of *Pos* using ROBDDs. In *Programming Languages: Implementation, Logics and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 1996.
3. R. Bagnara and P. Schachte. Factorizing Equivalent Variable Pairs in ROBDD-Based Implementations of Pos. In *Seventh International Conference on Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 1999.

4. N. Baker and H. Søndergaard. Definiteness Analysis for CLP($\mathcal{R}$). In *Australian Computer Science Conference*, pages 321–332, 1993.
5. P. Bigot, S. Debray, and K. Marriott. Understanding Finiteness Analysis using Abstract Interpretation. In *Joint International Conference and Symposium on Logic Programming*, pages 735–749. MIT Press, 1992.
6. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 108–124. Springer, 1996.
7. M. Codish. Efficient Goal Directed Bottom-up Evaluation of Logic Programs. *Journal of Logic Programming*, 38(3):355–370, 1999.
8. M. Codish and B. Demoen. Analysing Logic Programs using "prop"-ositional Logic Programs and a Magic Wand. *Journal of Logic Programming*, 25(3):249–274, 1995.
9. M. Codish, A. Heaton, A. King, M. Abo-Zaed, and P. Hill. Widening Positive Boolean Functions for Goal-dependent Groundness Analysis. Technical Report 12-98, Computing Laboratory, May 1998. http://www.cs.ukc.ac.uk/pubs/1998/589.
10. M. Codish, H. Søndergaard, and P. Stuckey. Sharing and Groundness Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 1999. To appear.
11. 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*, volume 714 of *Lecture Notes in Computer Science*, pages 75–91. Springer, 1993.
12. P. Dart. On Derived Dependencies and Connected Databases. *Journal of Logic Programming*, 11(1&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. B. De Backer and H. Beringer. A CLP Language Handling Disjunctions of Linear Constraints. In *International Conference on Logic Programming*, pages 550–563. MIT Press, 1993.
15. C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung*. PhD thesis, Universität des Saarlandes, 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 Transactions on Programming Languages and Systems*, 18(5):564–614, 1996.
17. M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–366, 1992.
18. A. King, J.-G. Smaus, and P. Hill. Quotienting Share for Dependency Analysis. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 1999.
19. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
20. G. Plotkin. A Note on Inductive Generalisation. *Machine Intelligence*, 5:153–163, 1970.
21. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain *Prop*. *Journal of Logic Programming*, 23(3):237–278, 1995.
22. J. Wunderwald. Memoing Evaluation by Source-to-Source Transformation. In *Logic Program Synthesis and Transformation*, Lecture Notes in Computer Science, pages 17–32. Springer, 1995. 1048.