Quotienting *Share* for Dependency Analysis

Andy King*, Jan-Georg Smaus* and Pat Hill**

*University of Kent at Canterbury, Canterbury, CT2 7NF, UK. amk@ukc.ac.uk, jgs5@ukc.ac.uk

**School of Computer Studies, University of Leeds, Leeds, LS2 9JT, UK. hill@scs.leeds.ac.uk

Abstract. Def, the domain of definite Boolean functions, expresses (sure) dependencies between the program variables of, say, a constraint program. $Share$, on the other hand, captures the (possible) variable sharing between the variables of a logic program. The connection between these domains has been explored in the domain comparison and decomposition literature. We develop this link further and show how the *meet* (as well as the *join*) of Def can be modelled with efficient (quadratic) operations on $Share$. Further, we show how by compressing and widening $Share$ and by rescheduling *meet* operations, we can construct a dependency analysis that is surprisingly fast and precise, and comes with time- and space- performance guarantees. Unlike some other approaches, our analysis can be coded straightforwardly in Prolog.

Keywords. (Constraint) logic programs, abstract interpretation, dataflow analysis, dependency analysis, definite Boolean functions, widening.

1 Introduction

Many analyses for logic programs, constraint logic programs and deductive databases use Boolean functions to express dependencies between program variables. In groundness analysis [2, 4, 10, 20, 26], the formula $x \wedge (y \leftarrow z)$ 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. Other useful properties like definiteness [5, 21], strictness [19], and finiteness [6] can be also expressed and inferred with Boolean functions. Different classes of Boolean functions have different degrees of expressiveness. For example, Pos, the class of positive propositional formulae, has the condensing [1] property and is rich enough for goal-independent analysis. Def, the class of definite positive propositional formulae, is less expressive [1] but has been proposed for goal-dependent analysis of constraint programs [21].

The objective behind this work was to construct a goal-dependent groundness (and definiteness) analysis for logic (and constraint) programs, that was fast and precise enough to be practical, maintainable and easy to integrate into a Prolog compiler. Binary Decision Diagrams (BDD's) [7] (and their derivatives like ROBDD's) are the popular choice for implementing a dependency analysis [1, 2, 4, 20, 26]. These are essentially directed acyclic graphs in which identical subgraphs are collapsed together. BDD operations require pointer manipulation and dynamic hashing [20] and thus BDD-based Pos analyses are usually implemented in C [1, 2, 4, 26]. Fecht [20] describes a notable exception that is coded in ML. The advantage of using ML is that it is more declarative than C and therefore easier to maintain. The disadvantage is that it impedes integration into a Prolog compiler [25]. The ideal, we believe, is to implement a dependency analyser in ISO Prolog. The problem, then, is essentially one of performance.

Our contribution to solving this problem is as follows: In terms of precision, we provide the first systematic precision experiments that compare *Pos* and *Def* for goal-dependent groundness (and definiteness) analysis. We found that *Def* was as precise as *Pos* for all our realistic Prolog and CLP($\mathcal{R}$) benchmarks. We build on this and demonstrate how *Def* can be implemented efficiently and coded succinctly in Prolog. Our starting point is the work of Cortesi *et al* [15, 16] that shows that *Share*, which is a domain whose elements are sets of sets of variables, can be used to encode *Def*. We develop this to show:

- how the *meet* and *join* operations of *Def* can be computed straightforwardly based on this encoding, without the closure operation of *Share* [22] that has a worst-case exponential complexity;
- how an operation (that we call compression) aids fixpoint detection;
- how *meet* operations can be rescheduled to improve efficiency;
- how widening can be applied to ensure that both the time-complexity of the analysis (the number of iterations) and the space-complexity (the number of sets of variables), grows linearly in the size of the program;
- that the speed of our analysis compares surprisingly well against state-of-the-art BDD-based *Pos* analysers [4, 20].

The rest of the paper is structured as follows. Section 2 surveys the necessary preliminaries. Section 3 recalls the relation between *Share* and *Def* and is included so that the paper is self-contained. Section 4 shows how the *meet* and *join* operations of *Def* can be computed efficiently using a *Share* based representation. Section 5 introduces compression and *meet* scheduling whereas Section 6 discusses widening. Section 7 describes the implementation. Section 8 reviews the related work, and finally Section 9 presents our conclusions.

2 Preliminaries

In this section, we introduce some notation and recall the definitions of Boolean functions and the domain *Share*. For a set S, $|S|$ denotes the cardinality and $\wp(S)$ the powerset of S. *Var* denotes a denumerable set (universe) of variables and $X \subset \mathit{Var}$ denotes a finite set of variables; the set of variables occurring in a syntactic object o is denoted by $\mathit{var}(o)$; the set of all idempotent substitutions is denoted by *Sub*; and *Bool* is defined to be $\{true, false\}$.

If $(S, \preceq)$ is a poset with top and bottom elements, and a *meet sqcap* and *join* $\sqcup$, then the 4-tuple $\langle S, \preceq, \sqcap, \sqcup \rangle$ denotes the corresponding lattice. A map $g : L \to K$, where L and K are lattices, is a homomorphism iff g is *join*-preserving and *meet*-preserving, that is, $g(a \sqcup b) = g(a) \sqcup g(b)$ and $g(a \sqcap b) = g(a) \sqcap g(b)$ for all $a, b \in L$. An isomorphism is a bijective homomorphism.

2.1 Boolean Functions

A Boolean function is a function $f : Bool^n \to Bool$ 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 $Bool_X$. We use Boolean functions and propositional formulae interchangeably without worrying about the distinction [1]. We follow the convention of identifying a truth assignment with the set of variables that it maps to *true*.

Definition 1 $model_X$**.** The (bijective) map $model_X : Bool_X \to \wp(\wp(X))$ is defined by: $model_X(f) = \{M \subseteq X \mid (\wedge M) \wedge (\neg \vee X \backslash M) \models f\}$.

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

Definition 2 Pos_X**,** Def_X**,** Mon_X**.** Pos_X is the set of positive Boolean functions over X. A function f is positive iff $X \in model_X(f)$. Def_X is the set of positive functions over X that are definite. A function f is definite iff $M \cap M' \in model_X(f)$ for all $M, M' \in model_X(f)$. Mon_X is the set of monotonic Boolean functions over X. A function f is monotonic iff $M \in model_X(f)$ implies $M' \in model_X(f)$ for all M' such that $M \subseteq M' \subseteq X$.

Note that $Def_X \subseteq Pos_X$ and $Mon_X \not\subseteq Pos_X$. It is possible to show that each $f \in Def_X$ is equivalent to a conjunction of definite (propositional) clauses, that is, $f = \wedge_{i=1}^{n}(y_i \leftarrow \wedge Y_i)$ [18].

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

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

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.

Defining $f_1 \dot{\vee} f_2 = \wedge\{f \in Def_X \mid f_1 \models f \wedge f_2 \models f\}$, the 4-tuple $\langle Def_X, \models, \wedge, \dot{\vee}\rangle$ 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, $\exists x.f = f[x \mapsto true] \vee f[x \mapsto false]$. Note that $\exists x.f \in Def_X$ if $f \in Def_X$ [1].

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

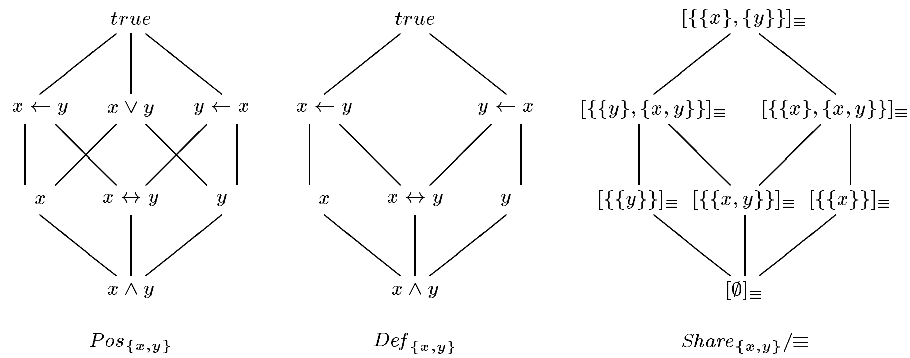

Fig. 1. Hasse diagrams

The maximum number of iterations of a fixpoint analysis relates to the length of the longest ascending chain in the underlying domain. For Pos_X, it is well-known that the longest chain has length $2^n - 1$ where $|X| = n$. It is less well-known that the same holds for Def_X.

Proposition 3. Let $|X| = n$. Let $f_1 \models f_2 \ldots \models f_k$ be a maximal strictly ascending chain where $f_i \in Def_X$ for all $i \in \{1, \ldots, k\}$. Then $k = 2^n$. ▮

2.2 Sharing Abstractions

For completeness, we introduce the basic ideas behind the *Share* domain [22]. This domain traces the possible variable sharing behaviour of a logic program. Two variables share if they are bound to terms that contain a common variable.

Definition 4 $Share_X$**.** $Share_X = \wp(\wp(X) \setminus \{\emptyset\})$. ▮

Thus we have the finite lattice $\langle Share_X, \subseteq, \cap, \cup \rangle$. The top element is $\wp(X) \setminus \{\emptyset\}$ and the bottom element is $\emptyset$.

Definition 5 α^{sh}_X**,** γ^{sh}_X**.** The abstraction map $\alpha^{sh}_X : \wp(Sub) \to Share_X$ is defined as $\alpha^{sh}_X(\Theta) = \{occ(\theta, v) \cap X \mid \theta \in \Theta \wedge v \in Var\} \setminus \{\emptyset\}$ where $occ(\theta, v) = \{x \in Var \mid v \in var(\theta(x))\}$. The concretisation map $\gamma^{sh}_X : Share_X \to \wp(Sub)$ is defined as $\gamma^{sh}_X(S) = \{\theta \in Sub \mid \alpha^{sh}_X(\{\theta\}) \subseteq S\}$. ▮

To streamline the theory and reduce the size of abstractions, the empty set is never included in a share set. However there is some loss of information. That is, if every element of Θ maps every element of X to a ground term then $\alpha^{sh}_X(\Theta) = \{\emptyset\} \setminus \{\emptyset\} = \emptyset = \alpha^{sh}_X(\emptyset)$. Thus α^{sh}_X (and hence γ^{sh}_X) cannot distinguish between a set of ground substitutions and the empty set. In practice, the empty set only arises when a computation fails and this would normally be flagged elsewhere in the analyser [9].

Example 4. Let $X = \{x, y, z\}$ and consider abstracting ⓐ $x = f(y, z)$ ⓑ where at program point ⓐ, no variable in X is ground or shares with any other element of X. The bindings on X, for example, could be θ_a or ϑ_a as given below. Then the bindings at ⓑ would be θ_b or ϑ_b, respectively.

$$\begin{array}{ll} \theta_a = \{y \mapsto g(u), z \mapsto v\} & \theta_b = \{x \mapsto f(g(u), v), y \mapsto g(u), z \mapsto v\} \\ \vartheta_a = \{x \mapsto f(u, u)\} & \vartheta_b = \{x \mapsto f(y, y), z \mapsto y\} \end{array}$$

The abstraction $S_a = \{\{x\}, \{y\}, \{z\}\}$ describes θ_a, that is $\theta_a \in \gamma_X^{sh}(S_a)$, since $occ(\theta_a, x) = \{x\}$, $occ(\theta_a, u) = \{u, y\}$, $occ(\theta_a, v) = \{v, z\}$ and $occ(\theta_a, y) = occ(\theta_a, z) = \emptyset$. Similarly $\vartheta_a \in \gamma_X^{sh}(S_a)$. The abstract unification operation of Jacobs and Langen [22] will compute the abstraction $S_b = \{\{x, y\}, \{x, z\}, \{x, y, z\}\}$ for the program point ⓑ. A safety result of Jacobs and Langen [22] asserts that $\theta_b, \vartheta_b \in \gamma_X^{sh}(S_b)$. Indeed, we see that $\theta_b \in \gamma_X^{sh}(S_b)$ since $occ(\theta_b, u) = \{u, x, y\}$, $occ(\theta_b, v) = \{v, x, z\}$, and $occ(\theta_b, x) = occ(\theta_b, y) = occ(\theta_b, z) = \emptyset$. The reader is encouraged to verify that $\vartheta_b \in \gamma_X^{sh}(S_b)$. ▮

3 Quotienting $Share_X$ to obtain Def_X

In this section we construct a homomorphism from $Share_X$ to Def_X. We recall the well-known connection between $Share_X$ and Def_X [13, 14, 15, 16]. For the elements of $Share_X$, we define an abstraction α_X which interprets a sharing abstraction as representing a set of models and hence a Boolean function.

Definition 6 α_X**.** The (abstraction) map $\alpha_X : Share_X \to Def_X$ is defined as follows: $\alpha_X(S) = model_X^{-1}(\{X \setminus (\cup S') \mid S' \subseteq S\})$. ▮

The definition of α_X is essentially that of α of Cortesi *et al* [14, Section 8.4], adapted to our definition of $Share_X$. α_X is well-defined, that is, $\alpha_X(S) \in Def_X$ for all $S \in Share_X$. First, since $X \in model_X(\alpha_X(S))$, it follows that $\alpha_X(S) \in Pos_X$. Secondly, if $M_1, M_2 \in model_X(\alpha_X(S))$ then $M_i = X \setminus (\cup S_i)$ where $S_i \subseteq S$ $(i = 1, 2)$. Clearly $S_1 \cup S_2 \subseteq S$. As $M_1 \cap M_2 = X \setminus (\cup(S_1 \cup S_2))$, it follows that $M_1 \cap M_2 \in model_X(\alpha_X(S))$.

Lemma 7. α_X is surjective. ▮

However, α_X is not injective, and thus it is a strict abstraction of $Share_X$. As an example, consider $X = \{x, y\}$, $S_1 = \{\{x\}, \{y\}\}$ and $S_2 = S_1 \cup \{\{x, y\}\}$. Then $\alpha_X(S_1) = model_X{}^{-1}(\{\emptyset, \{x\}, \{y\}, \{x, y\}\}) = \alpha_X(S_2)$ but $S_1 \neq S_2$.

Example 5. Let $X = \{x, y, z\}$ and $S = \{G_1, G_2, G_3\}$ where $G_1 = \{x\}$, $G_2 = \{y, z\}$ and $G_3 = \{z\}$. The table illustrates how $\alpha_X(S)$ can be computed by enumerating $\cup S'$ and $X \setminus (\cup S')$ for all $S' \subseteq S$.

S'	$\cup S'$	$X \setminus (\cup S')$	S'	$\cup S'$	$X \setminus (\cup S')$
$\emptyset$	$\emptyset$	$\{x, y, z\}$	$\{G_3\}$	$\{z\}$	$\{x, y\}$
$\{G_1\}$	$\{x\}$	$\{y, z\}$	$\{G_1, G_3\}$	$\{x, z\}$	$\{y\}$
$\{G_2\}$	$\{y, z\}$	$\{x\}$	$\{G_2, G_3\}$	$\{y, z\}$	$\{x\}$
$\{G_1, G_2\}$	$\{x, y, z\}$	$\emptyset$	$\{G_1, G_2, G_3\}$	$\{x, y, z\}$	$\emptyset$

Thus $\alpha_X(S) = model_X{}^{-1}(\{\emptyset, \{x\}, \{y\}, \{x, y\}, \{y, z\}, \{x, y, z\}\}) = (y \leftarrow z)$. The reader is encouraged to verify that $\alpha_X(\emptyset) = \wedge X$ and $\alpha_X(\{\{x\} \mid x \in X\}) = \mathit{true}$. ▮

It is perhaps easier to interpret an abstraction of Share_X as definite Boolean functions by using the $\mathcal{C} : \mathit{Share}_X \to \mathit{Def}_X$ abstraction map of Cortesi *et al* [15, 16]. $\mathcal{C}$ can be expressed particularly succinctly using the auxiliary operation *rel* which, given a set of variables G and an $S \in \mathit{Share}_X$, selects the subset of S which is relevant to the variables of G.

Definition 8 ***rel*.** The map $\mathit{rel} : \wp(X) \times \mathit{Share}_X \to \mathit{Share}_X$ is defined by:

$$\mathit{rel}(Y, S) = \{G \in S \mid G \cap Y \neq \emptyset\}$$ ▮

Definition 9. The map $\mathcal{C} : \mathit{Share}_X \to \mathit{Def}_X$ is defined by $\mathcal{C}(S) = \wedge F$ where

$$F = \{y \leftarrow \wedge Y \mid y \in X \ \wedge \ Y \subseteq X \setminus \{y\} \ \wedge \ \mathit{rel}(\{y\}, S) \subseteq \mathit{rel}(Y, S)\}.$$ ▮

F is defined with $Y \subseteq X \setminus \{y\}$ rather than $Y \subseteq X$ to keep its size manageable.

Example 6. Consider again Example 5. The set of $Y \subseteq X \setminus \{x\}$ such that $\mathit{rel}(\{x\}, S) \subseteq \mathit{rel}(Y, S)$ is $\{\{x\}, \{x, y\}, \{x, z\}, \{x, y, z\}\}$. Likewise, set of $Y \subseteq X \setminus \{y\}$ such that $\mathit{rel}(\{y\}, S) \subseteq \mathit{rel}(Y, S)$ is $\{\{y\}, \{z\}, \{x, z\}, \{x, y\}, \{y, z\}, \{x, y, z\}\}$. Finally, the set of $Y \subseteq X \setminus \{z\}$ such that $\mathit{rel}(\{z\}, S) \subseteq \mathit{rel}(Y, S)$ is $\{\{z\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$. Thus $\mathcal{C}(S) = (y \leftarrow z)$. ▮

The following proposition asserts the equivalence of $\mathcal{C}$ and α_X. It is proven by Cortesi *et al* [14], albeit for slightly different definitions. Modifying their proof to our definitions is straightforward.

Proposition 10. $\mathcal{C} = \alpha_X$. ▮

By defining $S \equiv S'$ iff $\alpha_X(S) = \alpha_X(S')$, α_X induces an equivalence relation on Share_X which quotients Share_X. Using the closure under union operation of Jacobs and Langen [22], we obtain a useful lemma about these equivalence classes.

Definition 11. Let $S \in \mathit{Share}_X$. Then the closure under union $S^\star$ of S is defined by: $S^\star = \{\cup S' \mid S' \subseteq S\} \setminus \{\emptyset\}$. ▮

Note that closure under union is exponential.

Lemma 12. Let $S_1, S_2 \in \mathit{Share}_X$. Then $S_1{}^\star \equiv S_1$ and $S_1 \equiv S_2$ iff $S_1^\star = S_2^\star$ ▮

We lift α_X to $\alpha_X : \mathit{Share}_X/\!\equiv \to \mathit{Def}_X$ by defining $\alpha_X([S]_\equiv) = \alpha_X(S)$. Since $\alpha_X : \mathit{Share}_X \to \mathit{Def}_X$ is surjective it follows that $\alpha_X : \mathit{Share}_X/\!\equiv \to \mathit{Def}_X$ is bijective. We now define, for the the operations $\models$, $\dot\vee$ and $\wedge$ on Def_X, analogous operations $\sqsubseteq, \sqcup$ and $\sqcap$ on $\mathit{Share}_X/\!\equiv$.

Definition 13 $\sqsubseteq$**,** $\sqcup$**,** $\sqcap$**.**

$$\begin{array}{lcl} [S_1]_\equiv \sqsubseteq [S_2]_\equiv & \leftrightarrow & \alpha_X([S_1]_\equiv) \models \alpha_X([S_2]_\equiv) \\ [S_1]_\equiv \sqcup [S_2]_\equiv & = & \alpha_X{}^{-1}(\alpha_X([S_1]_\equiv) \ \dot\vee \ \alpha_X([S_2]_\equiv)) \\ [S_1]_\equiv \sqcap [S_2]_\equiv & = & \alpha_X{}^{-1}(\alpha_X([S_1]_\equiv) \wedge \alpha_X([S_2]_\equiv)) \end{array}$$ ▮

Proposition 14. $\langle \mathit{Share}_X/\!\equiv, \sqsubseteq, \sqcap, \sqcup \rangle$ is a finite lattice. ▮

It follows by construction that α_X is an isomorphism. For the dyadic case, the isomorphism is illustrated in Fig. 1.

4 Computing the *join* and *meet* within *Share*

In this section we show how the *meet* (as well as the *join*) of Def_X can be computed with $Share_X/\equiv$ via the isomorphism. It is not obvious from the definition of $\dot{\vee}$ how $f_1 \dot{\vee} f_2$ is computed, and it turns out that f_1 and f_2 must be put into (orthogonal) reduced monotonic body form [1]. In contrast, it is well-known [15, 16] that with the *Share* representation, *join* basically reduces to set union.

Proposition 15. $[S_1]_\equiv \sqcup [S_2]_\equiv = [S_1 \cup S_2]_\equiv$ ∎

Example 7. Consider calculating $[S_1]_\equiv \sqcup [S_2]_\equiv$ where $X = \{w, x, y, z\}$, $S_1 = \{\{w, x, y\}, \{x, y\}, \{y\}, \{z\}\}$ and $S_2 = \{\{w, z\}, \{x\}, \{y\}, \{z\}\}$. Note that $\alpha_X(S_1) = (w \leftarrow x) \wedge (x \leftarrow y)$ and $\alpha_X(S_2) = w \leftarrow z$. Then $\alpha_X(S_1 \cup S_2) = \alpha_X(\{\{w, x, y\}, \{w, z\}, \{x\}, \{x, y\}, \{y\}, \{z\}\}) = (w \leftarrow (x \wedge z)) \wedge (w \leftarrow (y \wedge z))$ as required. ∎

The challenge is in defining a computationally efficient *meet*. This is defined in terms of a map *iff* which, in turn, is defined in terms of the binary-union operation of Jacobs and Langen [22]. We follow Cortesi *et al* [16] and denote binary union as $\otimes$.

Definition 16 binary-union, $\otimes$. The map $\otimes : {Share_X}^2 \to Share_X$ is defined by: $S_1 \otimes S_2 = \{G_1 \cup G_2 \mid G_1 \in S_1 \wedge G_2 \in S_2\}$. ∎

The *if* and *iff* maps defined below are similar to the classical abstract unification operation of Jacobs and Langen [22]. Their interpretation, however, is that given variable sets Y_1 and Y_2 and an abstraction S such that $\alpha_X(S) = f$, *iff* and *if* compute new abstractions that represent $f \wedge (\wedge Y_1 \leftrightarrow \wedge Y_2)$ and $f \wedge (\wedge Y_1 \leftarrow \wedge Y_2)$.

Definition 17. The two maps $iff : \wp(X) \times \wp(X) \times Share_X \to Share_X$ and $if : \wp(X) \times \wp(X) \times Share_X \to Share_X$ are defined by: $iff(Y_1, Y_2, S) = (S \setminus (S_1 \cup S_2)) \cup (S_1 \otimes S_2)$ and $if(Y_1, Y_2, S) = (S \setminus S_1) \cup (S_1 \otimes S_2)$ where $S_1 = rel(Y_1, S)$ and $S_2 = rel(Y_2, S)$. ∎

One important difference between *iff* and *if* on the one hand and the abstract unification algorithm of Jacobs and Langen [22] on the other hand is that *iff* and *if* involve no costly closure calculations that arise because of the transitivity of variable sharing. Consequently the complexity *iff* and *if* is not exponential in the number of variable sets in S, but quadratic. This is a similar efficiency gain to that obtained with the *Share* pair-sharing quotient of Bagnara *et al* [3].

Proposition 18. $\alpha_X(iff(Y_1, Y_2, S)) = \alpha_X(S) \wedge (\wedge Y_1 \leftrightarrow \wedge Y_2)$ ∎

Corollary 19. $\alpha_X(if(Y_1, Y_2, S)) = \alpha_X(S) \wedge (\wedge Y_1 \leftarrow \wedge Y_2)$ ∎

Even though $if(Y_1, Y_2, S)$ can be simulated with $iff({Y_1}', Y_2, S)$ where ${Y_1}' = Y_1 \cup Y_2$, it is cheaper to compute $rel(Y_1, S)$ than $rel({Y_1}', S)$. This is one reason why $if(Y_1, Y_2, S)$ is more efficient than $iff({Y_1}', Y_2, S)$. The map *if* is particularly useful in the analysis of constraint logic programs, where a constraint like $x = y + z$ is abstracted by $(x \leftarrow (y \wedge z)) \wedge (y \leftarrow (x \wedge z)) \wedge (z \leftarrow (x \wedge y))$.

Projection is an important component of a *Def* analysis within itself [21]. For completeness, we state its correctness as a proposition.

Definition 20 projection $\overline{\exists}$. The map $\overline{\exists} : \wp(X) \times Share_X \to Share_X$ is defined by: $\overline{\exists}Y.S = \{G \cap Y \mid G \in S\} \setminus \{\emptyset\}$.

Proposition 21. If $Y \subseteq X$ then $\exists(X \setminus Y).\alpha_X([S]_\equiv) = \alpha_Y([\overline{\exists}Y.S]_\equiv)$.

Finally, Theorem 22 shows how *meet* can be computed with a sequence of *iff* calculations.

Theorem 22. $[S_1]_\equiv \sqcap [S_2]_\equiv = [\overline{\exists}X.S'_{n+1}]_\equiv$ where $X = \{x_1, \ldots, x_n\}$, $S'_1 = \rho(S_1) \cup S_2$, $S'_{j+1} = \mathit{iff}(\{\rho(x_j)\}, \{x_j\}, S'_j)$ for $j \in \{1, \ldots, n\}$ and ρ is a renaming such that $\rho(X) \cap X = \emptyset$.

Note that $[S_1]_\equiv \sqcap [S_2]_\equiv$ could also be computed by $[S_1]_\equiv \sqcap [S_2]_\equiv = [{S_1}^\star \cap {S_2}^\star]_\equiv$. This, however, would be inefficient.

Example 8. Consider calculating $[S_1]_\equiv \sqcap [S_2]_\equiv$ where $X = \{w, x, y\}$, $S_1 = \{\{w, x\}, \{x\}, \{y\}\}$ and $S_2 = \{\{w\}, \{x, y\}, \{y\}\}$. Thus $\alpha_X(S_1) = w \leftarrow x$ and $\alpha_X(S_2) = x \leftarrow y$. If $\rho = \{w \mapsto w', x \mapsto x', y \mapsto y'\}$ then

$$\begin{array}{ll}
S'_1 = & \{\{w', x'\}, \{x'\}, \{y'\}, \{w\}, \{x, y\}, \{y\}\} \\
S'_2 = \mathit{iff}(\{w'\}, \{w\}, S'_1) = & \{\{w', x', w\}, \{x'\}, \{y'\}, \{x, y\}, \{y\}\} \\
S'_3 = \mathit{iff}(\{x'\}, \{x\}, S'_2) = & \{\{w', x', w, x, y\}, \{x', x, y\}, \{y'\}, \{y\}\} \\
S'_4 = \mathit{iff}(\{y'\}, \{y\}, S'_3) = & \{\{w', x', y', w, x, y\}, \{x', y', x, y\}, \{y', y\}\}
\end{array}$$

Thus $[S_1]_\equiv \sqcap [S_2]_\equiv = [\overline{\exists}X.{S_4}']_\equiv = \{\{w, x, y\}, \{x, y\}, \{y\}\}$. Observe that $\alpha_X([S_1]_\equiv \sqcap [S_2]_\equiv) = (w \leftarrow x) \wedge (x \leftarrow y)$ as required.

5 Representing equivalence classes and *meet* rescheduling

In our analysis, the functions f and f' would be represented by elements of $Share_X$, S and S', say. The fixpoint stability check, $f = f'$, amounts to checking whether $[S]_\equiv = [S']_\equiv$ which, in turn, reduces to deciding whether $\alpha_X(S) = \alpha_X(S')$. To make this test efficient we represent an equivalence class by its smallest representative and thus introduce a compression operator c.

Definition 23. $c : Share_X \to Share_X$ is defined by: $c(S) = \cap\{S' \mid S' \equiv S\}$.

The following proposition explains how $c(S)$ is actually computed.

Proposition 24. *Let* $n = |X|$. *Then* $c(S) = S_n$ *where* $S_1 = \{G \in S \mid |G| = 1\}$ *and* $S_{j+1} = S_j \cup \{G \in S \mid |G| = j + 1 \wedge G \notin S_j^\star\}$.

Trivially, if $S \equiv S'$, then $c(S) = c(S')$. From the proposition we also see that $S^\star = {S_n}^\star = c(S)^\star$ and hence if $c(S) = c(S')$ then $S^\star = c(S)^\star = c(S')^\star = S'^\star$ so that $S \equiv S'$ by Lemma 12. Hence $c(S) = c(S')$ iff $S \equiv S'$ and thus by testing whether $c(S) = c(S')$ we can check for the fixpoint condition $S \equiv S'$.

When computing $c(S)$ we can test whether $G \notin {S_j}^\star$ without actually computing ${S_j}^\star$ as follows. Suppose $S_j = \{G_1, \ldots, G_m\}$ and ${G_0}' = G$. Then compute ${G_i}' = {G_{i-1}}' \setminus G_i$ if $G_i \subseteq G$ and put ${G_i}' = {G_{i-1}}'$ otherwise. Then ${G_m}' = \emptyset$ iff $G \in {S_j}^\star$. Using this tactic we can compute $c(S)$ in quadratic time.

Projection can sometimes lead to abstractions that include redundant variable sets as is illustrated below.

Example 9. Consider $S = \{\{x\}, \{y\}, \{x, y, z\}\}$ which, incidentally, represents $\alpha_X(S) = (z \leftarrow x) \wedge (z \leftarrow y)$. Projecting onto $\{x, y\}$ like so $\overline{\exists}\{x, y\}.S = \{\{x\}, \{y\}, \{x, y\}\}$ introduces the set $\{x, y\}$, whereas $c(\overline{\exists}\{x, y\}.S) = \{\{x\}, \{y\}\}$. ▮

Compression is only applied to check for stability. In our framework, however, projection always precedes a stability check. For example, the answer pattern for a clause is obtained by projecting onto the head variables, and then a stability check is applied to see if other clauses need to be re-evaluated. Thus, in our framework, compression is applied after projection. Compression could be applied more widely though since, in general, $\mathit{iff}(Y_1, Y_2, c(S)) \neq c(\mathit{iff}(Y_1, Y_2, c(S)))$.

Example 10. Let $S = \{\{x\}, \{y, x\}, \{y, z\}, \{x, z\}\}$. Then $c(S) = S$ and $\mathit{iff}(\{y\}, \{z\}, S) = \{\{x\}, \{y, z\}, \{y, x, z\}\}$ but $c(\{\{x\}, \{y, z\}, \{y, x, z\}\}) = \{\{x\}, \{y, z\}\}$. ▮

In practice, however, the space saving yielded by $c(\mathit{iff}(Y_1, Y_2, S))$ over $\mathit{iff}(Y_1, Y_2, S)$ is usually small and not worth the effort of computing c.

Curiously, the efficiency of *meet* computations can often be significantly improved by introducing some redundancy into the representation. Specifically, a Boolean function is represented by a pair $\langle M, S\rangle$ where $M = X \setminus \mathit{var}(S)$. The pair $\langle M, S\rangle$ does not include any information that is not present in S: it simply flags those variables, M, that are ground (or definite). (This is reminiscent of the reactive ROBDD representation of Bagnara [2].) This is very useful in computing $[S_1]_\equiv \sqcap [S_2]_\equiv$ by the method prescribed in Theorem 22. Since *meet* is commutative, $[S_1]_\equiv \sqcap [S_2]_\equiv$ can be computed by the sequence $S'_1 = \rho(S_1) \cup S_2$, $S'_{j+1} = \mathit{iff}(\{\rho(x_{\pi(j)})\}, \{x_{\pi(j)}\}, S'_j)$ where π is a permutation on $\{1, \ldots, n\}$. The tactic is to choose a permutation with a maximal $m \in \{0, \ldots, n\}$ such that $(\rho(x_{\pi(1)}) \in M \vee x_{\pi(1)} \in M)$ $\ldots$ $(\rho(x_{\pi(m)}) \in M \vee x_{\pi(m)} \in M)$ where $M = M_1 \cup M_2$, $M_1 = X \setminus \mathit{var}(S_1)$ and $M_2 = X \setminus \mathit{var}(S_2)$. We call this technique *meet* rescheduling, and illustrate its usefulness in the following example.

Example 11. Consider $[S_1]_\equiv \sqcap [S_2]_\equiv$ where $X = \{x_1, x_2, x_3\}$, $S_1 = \{\{x_1, x_2\}\}$ and $S_2 = \{\{x_1\}, \{x_2, x_3\}\}$. Thus $\alpha_X(S_1) = (x_1 \leftrightarrow x_2) \wedge x_3$ and $\alpha_X(S_2) = (x_2 \leftrightarrow x_3)$. Also $M_1 = \{x_3\}$, $M_2 = \emptyset$ and thus $M = \{x_3\}$. If $\rho = \{x_1 \mapsto x_1', x_2 \mapsto x_2', x_3 \mapsto x_3'\}$ then scheduling naively and using $\pi = \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 2\}$ we obtain, respectively

$$
\begin{array}{ll}
S'_1 = \{\{x'_1, x'_2\}, \{x_1\}, \{x_2, x_3\}\} & S'_1 = \{\{x'_1, x'_2\}, \{x_1\}, \{x_2, x_3\}\} \\
S'_2 = \{\{x_1, x'_1, x'_2\}, \{x_2, x_3\}\} & S'_2 = \{\{x'_1, x'_2\}, \{x_1\}\} \\
S'_3 = \{\{x_1, x'_1, x'_2, x_2, x_3\}\} & S'_3 = \{\{x'_1, x_1, x'_2\}\} \\
S'_4 = \emptyset & S'_4 = \emptyset
\end{array}
$$

Note how the re-ordering π tends to reduce the size of the intermediate S'_i. ▮

A pair $\langle M, S\rangle$ representation is preferred to recomputing M prior to each *meet* because formulae typically occur as the operands of many *meet* operations. Thus M serves as a memo, avoiding unnecessary recomputation.

6 Widening

Apart from reducing the size of abstractions, it is also worthwhile to avoid generating large abstractions that can arise from the quadratic growth of $\mathit{iff}(Y_1, Y_2, S)$ and $\mathit{if}(Y_1, Y_2, S)$ stemming from $S_1 \otimes S_2$. However, if $|S| = n$, $|S_1| = n_1$, $|S_2| = n_2$ then $|\mathit{iff}(Y_1, Y_2, S)| \leq n + n_1 n_2 - (n_1 + n_2)$. Thus it is possible to detect that $|\mathit{iff}(Y_1, Y_2, S)|$ will definitely be small, say less than a threshold k, without computing $\mathit{iff}(Y_1, Y_2, S)$ itself. This leads to the following (widened) versions of *iff* and *if* that trade precision for efficiency.

Definition 25.

$$\mathit{iff}_k(Y_1, Y_2, S) = \begin{cases} \mathit{iff}(Y_1, Y_2, S) & \text{if } n + n_1 n_2 - (n_1 + n_2) \leq k \\ S & \text{otherwise} \end{cases} \qquad \mathit{if}_k(Y_1, Y_2, S) = \begin{cases} \mathit{if}(Y_1, Y_2, S) & \text{if } n + n_1 n_2 - n_1 \leq k \\ S & \text{otherwise} \end{cases}$$

where $S_1 = \mathit{rel}(Y_1, S)$, $S_2 = \mathit{rel}(Y_2, S)$, $|S| = n$, $|S_1| = n_1$ and $|S_2| = n_2$. ∎

This (space) widening ensures that at each stage of the analysis the size of an abstraction is kept smaller than k. In fact, since the size of the abstraction depends on the number of variables, k is defined as a multiple of the number of the variables in a clause. This is enough to ensure that, in our interpreter, our space usage grows linearly with the size of the program. A widened *meet* can be obtained by replacing each $\mathit{iff}(\{\rho(x_j)\}, \{x_j\}, S'_j)$ of Theorem 22 by $\mathit{iff}_k(\{\rho(x_j)\}, \{x_j\}, S'_j)$. (Interestingly, a widening for ROBDD's is described by Fecht [20] that combats the space problems that arise in the analysis of high arity predicates.)

Folklore [8] says that call and answer patterns rarely get updated more than 3–4 times. This is true for many small programs, but in chat_80.pl and aqua_c.pl we have observed patterns being updated 10–12 times. To bound the number of iterations that can occur, we widen abstractions if they are updated more than, say, 8 times. This (time) widening is defined by: $\triangle(S) = S' \cup \{\{x\} \mid x \in \mathit{var}(S) \setminus \mathit{var}(S')\}$ where $S' = \{G' \in S \mid \forall G \in S.(G \cap G' \neq \emptyset) \rightarrow (G' \subseteq G)\}$. Observe that $[S]_{\equiv} \sqsubseteq [\triangle(S)]_{\equiv}$ and that $\alpha_X(\triangle(S)) = (\wedge Y) \wedge (\wedge \{x \leftrightarrow y \mid G \in \triangle(S) \wedge x, y \in G\})$ where $Y = X \setminus \mathit{var}(\triangle(S))$. Formulae of this form occur in the $WPos$ domain of Codish *et al* [11] and thus have a maximal chain length that is linear in $|X|$. This ensures that the number of iterates will be linear in the sum of the arities of program predicates, and thus provides a time guarantee for a cautious compiler vendor.

7 Experimental work

To investigate whether a quadratic *meet*, *meet* rescheduling and widening are enough to obtain an efficient and scalable dependency analysis, we have implemented an analyser in Prolog as a simple meta-interpreter that uses induced magic-sets [9] and eager evaluation [27] 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 magic'ed clauses [9]. It also avoids the overhead of applying the magic set transformation. Eager evaluation [27] 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. Eager evaluation can involve more re-computation than semi-naive iteration but it has the advantages that (1) a (Δ-)set of recently updated atoms does not need to be represented; (2) eager evaluation performs a depth-first traversal of the call-graph so that information about strongly connected components (SCCs) of the call-graph is not as important as in semi-naive iteration. Thus we also avoid computing SCCs.

		fixpoint					precision			
file	abs	Con	Def_n	Def_r	Def_w	Pos	Con	Def	Def_w	Pos
disj_r.pl	0.13	0.06	0.13	0.09	0.08	0.03	38	97	97	97
scc1.pl	0.21	0.13	0.81	0.81	0.77	0.37	44	89	89	89
tictactoe.pl	0.22	0.02	0.12	0.08	0.09	0.04	56	56	56	56
dialog.pl	0.13	0.03	0.1	0.07	0.07	0.04	46	70	70	70
ime_v2-2-1.pl	0.18	0.05	0.56	0.3	0.29	0.36	77	101	101	101
cs_r.pl	0.26	0.06	0.11	0.1	0.09	0.05	36	149	149	149
flatten.pl	0.17	0.03	0.76	0.36	0.35	1.62	26	27	27	27
conman.pl	0.2	0.0	0.01	0.02	0.01	0.01	6	6	6	6
unify.pl	0.21	0.04	0.87	0.45	0.44	59.06	69	70	70	70
bridge.clpr	0.34	0.01	0.06	0.03	0.03	0.04	24	24	24	24
neural.clpr	0.25	0.05	0.58	0.15	0.14	0.12	80	118	118	118
kalah.pl	0.22	0.1	0.12	0.13	0.12	0.04	91	199	199	199
bryant.pl	0.28	0.06	1.46	1.04	1.03	70.24	89	89	89	89
nbody.pl	0.33	0.05	8.44	0.33	0.32	1.03	83	109	109	109
sdda.pl	0.25	0.04	0.34	0.38	0.38	3.33	17	17	17	17
peep.pl	0.51	0.04	0.37	0.3	0.29	0.87	8	10	10	10
boyer.pl	0.32	0.03	0.11	0.18	0.19	0.15	3	3	3	3
read.pl	0.38	0.05	0.81	0.42	0.4	1.15	90	99	99	99
qplan.pl	0.36	0.12	1.77	1.59	1.56	63.8	42	49	49	49
reducer.pl	0.31	0.04	∞	∞	0.93	∞	36	–	41	–
press.pl	0.36	0.17	11.6	4.23	1.26	2.29	45	53	53	53
asm.pl	0.51	0.07	0.33	0.39	0.43	0.23	86	87	87	87
parser_dcg.pl	0.39	0.08	0.53	0.58	0.55	0.97	25	41	41	41
trs.pl	0.52	0.09	3.08	2.51	2.48	∞	13	13	13	∞
dbqas.pl	0.36	0.02	0.31	0.09	0.09	0.23	36	43	43	43
ann.pl	0.41	0.09	1.9	1.27	0.94	1.99	73	73	73	73
nand.pl	0.49	0.62	0.67	0.39	0.39	0.16	123	402	402	402
simple_analyzer.pl	0.38	0.08	2.37	0.74	0.72	∞	88	89	89	–
sim.pl	0.76	0.18	3.62	2.51	2.46	∞	81	100	100	–
ili.pl	0.61	0.13	∞	∞	1.69	19.16	4	–	4	4
lnprolog.pl	0.41	0.16	0.37	0.53	0.5	0.23	54	143	143	143
rubik.pl	0.79	0.2	2.0	1.96	1.93	∞	153	160	160	–
strips.pl	0.79	0.04	0.17	0.16	0.16	0.06	144	144	144	144
peval.pl	0.68	0.08	1.92	1.49	1.06	9.34	27	27	27	27
sim_v5-2.pl	0.86	0.11	0.48	0.55	0.54	0.49	100	101	101	101
chat_parser.pl	1.09	0.62	4.27	3.88	3.52	∞	444	505	505	–
aircraft.pl	2.01	8.56	1.34	1.33	1.3	0.35	228	687	687	687
essln.pl	1.49	0.25	2.76	1.43	1.39	17.44	103	155	155	155
chat_80.pl	4.63	1.23	12.89	9.99	9.82	∞	457	839	839	–
aqua_c.pl	12.17	7.07	∞	∞	69.56	∞	1087	–	1227	–

The table summarises our experimental results for applying Def to some of the largest Prolog and CLP($\mathcal{R}$) benchmark programs that we could find on the WWW. The programs are ordered by size, where size is measured in terms of the number of (distinct abstract) clauses. To assess the precision of the Def analysis, we have implemented a standard Pos analysis following the technique of Codish and Demoen [10]. Ideally our Def analysis should match its precision. We have also modified this analysis to obtain a Con analysis [23]. Ideally our Def analysis should significantly improve on its precision, since otherwise neither Def or Pos are worthwhile! For completeness, we have included the timings for Pos and Con, but we are primarily concerned with precision. Our Pos analysis is not state-of-the-art. The *abs* column give the time for parsing the files and abstracting them, that is, replacing built-ins, like `arg(X, T, S)`, with formulae, like $\mathtt{X} \wedge (\mathtt{S} \leftarrow \mathtt{T})$. This overhead is the same for all the analyses. The *fixpoint* columns gives the time to compute the fixpoint. Def_n is a naive implementation of our analysis (that took two person weeks to construct) which applies compression but not *meet* rescheduling and widening; Def_r additionally applies *meet* rescheduling; and Def_w applies compression, *meet* rescheduling and widening. The Def_r and Def_w analysers were developed together and took an additional 4 days to construct. The code for Def_n, Def_r and Def_w meta-interpreters (including all the set manipulation utilities) is less than 700 clauses. We widen for time at iteration 8 and widen for space when the number of variable sets is more than 16 times the number of variables in a clause. Times are in seconds and ∞ indicates that the fixpoint calculation timed out after two minutes. The timings were carried out on an Sun-20 SuperSparc with 64 MByte to match the architecture of Fecht [20]. The analysers were coded in SICStus 3#5 and compiled to naive code. The *precision* columns give the total number of ground arguments in the call and answer patterns: this is an absolute measure which reflects the usefulness of the analysis for code optimisation. The precision figures for Def_n and Def_r are the same and given in column Def.

The experimental results indicate that Def_w has good scaling behaviour. This is the crucial point. Put simply, there are no programs for which Pos terminates within two minutes and Def_w does not (although Pos is sometimes faster). Usually *meet* rescheduling gives a speedup and sometimes this speedup is very dramatic. 10% of the programs, however, run slower with *meet* rescheduling. This typically occurs in programs with very few ground arguments where the effort of rescheduling in not repaid by a reduction in the size of sharing abstractions. Widening seems to be crucial for scalability as is illustrated by reducer, ili and aqua_c. Widening, in fact, is rarely applied. It is crucial for efficiency though because, just one large sharing abstraction can have a disastrous impact on performance. (This also suggests that widening is necessary in the pair-sharing quotient of *Share* [3].)

Since our machine matches that of Fecht [20] we can also compare the speed of our Def analyser to the BDD-based Pos GENA analyser [20]. This the one of the fastest (perhaps the fastest) Pos analysis that is described in the literature. With the sophisticated CallWDFS [20] framework, ann.pl takes 0.18 s, nand.pl takes 0.31 s, chat_80.pl takes 4.29 s, and aqua_c.pl takes 28.54 s. Since Fecht [20] does not give processor details for his Sparc-20, we have run our experiments on the slowest 50MHz model that was manufactured. His machine could well be almost twice as fast. Even though our framework is not semi-naive, we are (at most) 2–4 times as slow as GENA. Furthermore, to perform a comparison against China instantiated with Pos [4], Bagnara has run Def_n and China on a Pentium 200MHz PC with 64 MByte of memory. On trs.pl and chat_80.pl Def_n

take 3.17 s and 12.59 s respectively running interpreted SICStus 3#6 bytecode. CHINA takes 2.94 s and 6.24 s respectively. It seems reasonable to assume that with Def_w on the same PC, trs.pl and chat_80.pl would take $3.17 \times \frac{2.48}{3.08} \approx 2.55$ s and $12.59 \times \frac{9.82}{12.89} \approx 9.59$ s. This performance gap for chat_80.pl would be closed if naive code assembly was available for the PC. To summarise, the experimental results are very encouraging and despite the simplicity of the interpreter, our Def_w analysis appears to be fast, precise and scalable and, of course, can be implemented easily in Prolog.

8 Related work

Cortesi *et al* [15] first pointed out that *Share* expresses the groundness dependencies of Def. Quotienting was introduced by Cortesi *et al* [16] as a systematic way of obtaining the reference domain of [15]. Like Bagnara *et al* [3], we do not fully adhere to the quotienting terminology and methodology of Cortesi *et al* [15] but rather follow the standard convention [17] of inducing an equivalence relation ($\equiv$) from an abstraction map (α_X). Also, Lemma 6.2 of [16] can be interpreted as a way of computing the *meet* in *Def* with the classic abstract unification of Jacobs and Langen [22]. We take this further and show how the *meet* can be computed without exponential time closure operations.

Bagnara *et al* [3] point out that *Share* includes redundant sharing information with respect to pair-sharing. This work is related to ours in that our domain may be viewed as a further quotient of the pair-sharing domain. However, widening has not been explored for the pair-sharing domain although, we have shown that even for our simpler domain, that widening is crucial for scalability.

Armstrong *et al* [1] investigate various normal forms of Boolean functions and the relative precision of Pos and Def. C-based implementations of each representation are described. For the representations of Pos, it is concluded that ROBDD's give the fastest analysis. A specialised representation for Def, based on Dual Blake Canonical Form (DBCF), is found to be the fastest overall. For medium-sized programs it is several times faster than ROBDD's, and it is concluded that this is the representation likely to scale best for real 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. Our findings support this remark.

Bagnara and Schachte [4] develop the idea [2] that a hybrid implementation of ROBDD's 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 ROBDD's and thus is a useful implementation tactic. A comparison with our *Def* analysis has already been given. Fecht [20] compares his *Pos* analyser to that of Van Hentenryck *et al* [26] and concludes that his analyser is an order of magnitude faster. For reasons of space, the reader is referred to [20, pp. 305–307] for more details. Performance figures for another hybrid representation are given in [24]. We just observe that [4] and [20] are very good systems to measure against.

García de la Banda *et al* [21] represent *Def* functions in terms of a domain $\wp(X) \times \wp(X \times \wp(\wp(X)))$, so that the Herbrand constraint $x = f(y, z)$, 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 [1] in which a definite function is represented as $f = \wedge_{x \in X}(x \leftarrow M_x)$ where $M_x \in Mon_X$ and $x \notin M_x$. Orthogonality ensures that the *meet* is safe. Our work shows how this symbolic manipulation of definite function can be replaced with a simpler domain and simpler *join* and *meet* operations.

Corsini *et al* [12] describe how variants of *Pos* can be implemented using Toupie, a constraint language based on the μ-calculus. This BDD-based analysis appears to be at least five times as fast as [26] for success pattern analysis. Thus, if the analyser was extended with magic sets, say, it might lead to a very respectable goal-dependent analysis.

Codish and Demoen [10] describe a truth-table based implementation technique for *Pos* that would encode $(x_1 \leftrightarrow (x_2 \wedge x_3))$ as three tuples $\langle true, true, true\rangle$, $\langle false, \_, false\rangle$, $\langle false, false, \_\rangle$. A widening for this *Pos* analysis, $WPos$, is proposed by Codish *et al* [11] that amounts to a sub-domain of Def that cannot propagate dependencies of the form $y \leftrightarrow (y \wedge z)$, but only simple dependencies like $(x \leftrightarrow y)$. The main finding of Codish *et al* [11] is that $WPos$ looses only a small amount of precision for goal-dependent analysis of Prolog and CLP($\mathcal{R}$) programs.

9 Conclusions

We have developed the link between *Def* and *Share* to show how the *meet* of *Def* can be modelled with an efficient (quadratic) operation on *Share*. We have shown how to represent formulae succinctly with equivalence classes of sharing abstractions, and how formulae can be widened so as to avoid bad space behaviour. Putting these ideas together we have achieved a practical analysis that is fast, precise, robust and can be implemented easily in Prolog.

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*, pages 107–121. Springer-Verlag, 1996. LNCS 1140.
3. R. Bagnara, P. Hill, and E. Zaffanella. Set-Sharing is Redundant for Pair-Sharing. In *Static Analysis Symposium*, pages 53–67. Springer-Verlag, 1997. LNCS 1302.
4. 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*, Amazonia, Brazil, 1999. Springer-Verlag.
5. N. Baker and H. Søndergaard. Definiteness analysis for CLP($\mathcal{R}$). In *Australian Computer Science Conference*, pages 321–332, 1993.

6. 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.
7. R. Bryant. Symbolic Boolean manipulation with ordered binary-decision digrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
8. M. Codish. Worst Case Groundness Analysis. Technical report, Ben-Gurion University of the Negev, 1998. ftp://ftp.cs.bgu.ac.il/pub/people/mcodish/pathpos.ps.gz.
9. M. Codish. Efficient Goal Directed Bottom-up Evaluation of Logic Programs. *J. of Logic Programming (to appear)*, 1999.
10. M. Codish and B. Demoen. Analysing Logic Programs using "prop"-ositional Logic Programs and a Magic Wand. *J. Logic Programming*, 25(3):249–274, 1995.
11. 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.
12. 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 *In proceedings of Programming Language Implementation and Logic Programming*, pages 75–91. Springer-Verlag, 1993. LNCS 714.
13. A. Cortesi and G. Filé. Comparison and Design of Abstract Domains for Sharing Analysis. In *Italian Conference on Logic Programming*, pages 251–265, 1993.
14. A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in Abstract Interpretation. *ACM TOPLAS*, 19(1):7–47, January 1997.
15. A. Cortesi, G. Filé, and W. Winsborough. Comparison of Abstract Interpretations. In *International Conference on Automata, Languages and Programming*, pages 521–532. Springer-Verlag, 1992. LNCS 623.
16. A. Cortesi, G. Filé, and W. Winsborough. The Quotient of an Abstract Interpretation for Comparing Static Analyses. *Theoretical Computer Science*, 202(1–2):163–192, 1998.
17. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
18. P. Dart. On Derived Dependencies and Connected Databases. *J. Logic Programming*, 11(1&2):163–188, 1991.
19. 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.
20. C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung.* PhD thesis, Universität des Saarlandes, 1997. http://www.cs.uni-sb.de/RW/users/fecht/.
21. 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.
22. D. Jacobs and A. Langen. Static Analysis of Logic Programs. *J. Logic Programming*, 13(2 and 3):154–314, 1992.
23. N. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Ltd, 1987.
24. P. Schachte. *Precise and Efficient Static Analysis of Logic Programs.* PhD thesis, Department of Computer Science, The University of Melbourne, Melbourne, Australia, 1998. (to appear).
25. H. Seidel. Personal communication on Prolog compiler integration and SML. November 1997.
26. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain *Prop*. *J. Logic Programming*, 23(3):237–278, 1995.
27. J. Wunderwald. Memoing Evaluation by Source-to-Source Transformation. In *Logic Program Synthesis and Transformation*, pages 17–32. Springer, 1995. LNCS 1048.

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