Practical Dependency Analysis through a *Share* Quotient

Andy King[*] Pat Hill[†] Jan–Georg Smaus[*]

Abstract

Def, the domain of definite Boolean functions, expresses (sure) dependencies between the program variables of, say, a constraint program. The domain *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, quotienting 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*. We show how the connection leads to an attractive way of constructing a dependency analysis and how widening can make the approach practical.
Keywords. (Constraint) logic programs, abstract interpretation, data-flow 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, 9, 23], for example, 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 [4, 18], strictness [16, 20], and finiteness [7] can be also expressed and inferred with Boolean functions. Different classes of Boolean functions have different degrees of expressiveness and are therefore suited to different applications. For example, *Pos*, the class of positive propositional formulae, has the condensing [1] property and is rich enough for goal-independent analysis [16], whereas *Def*, the class of definite propositional formulae, is useful in goal-dependent analysis [18].

The domain *Share* [19], on the other hand, traces the possible variable sharing behaviour of a logic program. Two variables share if they are bound to terms that contain a common variable. The accuracy of *Share* stems, in part, from its ability to track groundness dependencies. This is because ground variables cannot share with other variables. Interestingly, *Share* can encode groundness dependencies to the accuracy of *Def* [12, 13]. In other words, *Share* is equivalent to the reduced product domain $Def \times Sharing^{+}$ [11], in which *Def* just traces groundness dependencies and $Sharing^{+}$ just traces possible sharing.

This paper develops the connection between *Share* and *Def* to show how the *meet* operation of *Def* (as well as the *join* [12, 13]) can be computed straightforwardly with a *Share* based set-of-sets representation. Previous studies of *Def* and *Share* have either focused on domain comparison (and quotienting) [12, 13] or domain decomposition [11] and, to the best of our knowledge, have not explored *Share* as a medium for efficient dependency analysis. In terms of expressiveness, our *Share* based *join* and *meet* operations are engineered to trace only sure dependency information. Possible sharing information is not preserved. The resulting *Def* analysis could be used for any kind of dependencies — it is not restricted to tracing groundness. In terms of efficiency, the closure operator of *Share* [19] is not required and, with widening, the size of the abstractions can be kept small. Like [3], our *Share* operations do not have a worst-case exponential complexity but unlike [3] the representation is, in addition, kept small.

[*]University of Kent at Canterbury, Canterbury, CT2 7NF, UK, {a.m.king, j.g.smaus}@ukc.ac.uk.
[†]School of Computer Studies, University of Leeds, Leeds, LS2 9JT, UK, hill@scs.leeds.ac.uk.

Thus the analysis does not suffer from the poor scaling behaviour that has dogged classical *Share* [17]. Our results do not contradict the equivalence problem for Def that is known to be in co-NP [1, Section 4]. This says that, in the worst-case, either the representation is of exponential size or the test for equivalence takes exponential time. We tackle this complexity problem by applying quadratric operations to a representation that is potentially exponential represent but limit the size of the representation through widening, albeit at a (neglible) loss of precision. Practically, the resulting dependency analysis reduces to simple operations on sets of sets of program variables. These operations can be coded straightforwardly in Prolog [21] to be integrated into a system like PLAI (or a CLP extension [18]). Furthermore, our domain operations are simpler than those described in [18] and, for example, do not require an expensive transitive closure operation.

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 Section 4 shows how the *meet* and *join* operations of Def can be computed efficiently using a *Share* based representation. Section 5 introduces a novel widening and a compression operator that helps stability detection and keeps the abstractions small, and Section 6 describes how the theory translates into an implementation. Section 7 reviews the related work, Section 8 discusses some directions for future work, and finally Section 9 presents some concluding remarks.

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 power set of S. Var denotes a denumerable set (universe) of variables and $X \subset Var$ denotes a totally ordered, finite set of variables; the set of variables occurring in a syntactic object o is denoted by $var(o)$; 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 *meet* and *join* operations $\sqcap$ and $\sqcup$, respectively, 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 homomorphism that is bijective.

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 2.1 [$model_X$] 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 2.1 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 $x \wedge y$, and $model_X(x \wedge y)$ $= \{\{x, y\}\}$. Similarly, $model_X(x \vee y) = \{\{x\}, \{y\}, \{x, y\}\}$. ▌

Definition 2.2 [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 $M \in model_X(f)$ implies $M' \in model_X(f)$ for all M' such that $M \subseteq M' \subseteq X$. ▌

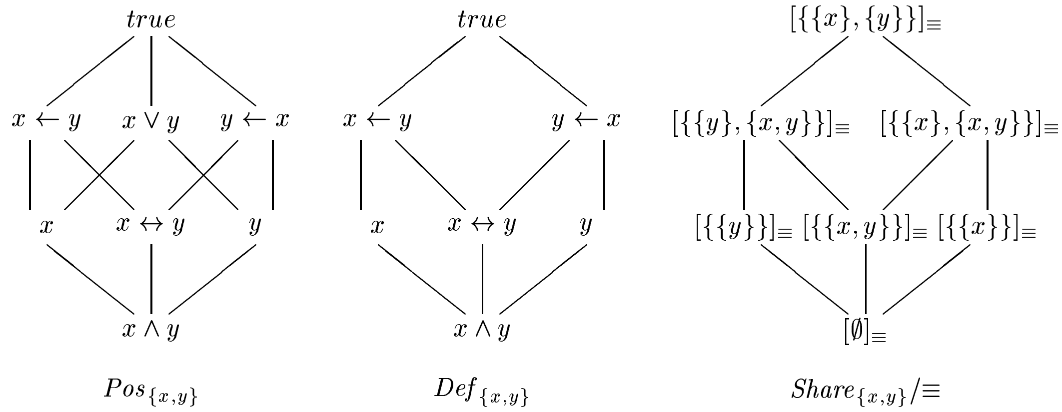

Figure 1: Hasse diagrams

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, $f = \wedge_{i=1}^{n}(y_i \leftarrow \wedge Y_i)$ [15].

Example 2.2 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 2.3 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 $x \dot{\vee} y = \wedge\{true\} = true \neq (x \vee y)$.

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 where $|X| = n$. It is less well-known that the same holds for Def_X.

Proposition 2.1 Let $|X| = n$. Let $f_1 \models f_2 \ldots \models f_k$ be a maximal 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 [19].

Definition 2.3 [$Share_X$] $Share_X = \wp(\wp(X) \setminus \{\emptyset\})$.

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 2.4 $[\alpha_X^{sh}, \gamma_X^{sh}]$ The abstraction map $\alpha_X^{sh} : \wp(Sub) \to Share_X$ is defined as $\alpha_X^{sh}(\Theta) = \{occ(\theta, v) \cap X \mid \theta \in \Theta \wedge v \in Var\} \setminus \{\emptyset\}$ where $occ(\theta, v) = \{x \in Var \mid v \in var(\theta(x))\}$. The concretisation map $\gamma_X^{sh} : Share_X \to \wp(Sub)$ is defined as $\gamma_X^{sh}(S) = \{\theta \in Sub \mid \alpha_X^{sh}(\{\theta\}) \subseteq S\}$. ▌

Note that to streamline the theory and, importantly, reduce the size of the 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 $\alpha_X^{sh}(\Theta) = \{\emptyset\} \setminus \{\emptyset\} = \emptyset = \alpha_X^{sh}(\emptyset)$. Thus α_X^{sh} (and hence γ_X^{sh}) 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 [21].

Example 2.4 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 [19] will compute the abstraction $S_b = \{\{x, y\}, \{x, z\}, \{x, y, z\}\}$ for the program point ⓑ. A safety result in [19] asserts that $\theta_b, \vartheta_b \in \gamma_X^{sh}(S_b)$. Indeed, we see that $\theta_b \in \gamma_X^{sh}(S_b)$. $occ(\theta_b, u) = \{u, x, y\}$, $occ(\theta_b, v) = \{v, x, z\}$, and $occ(\theta_b, x) = occ(\theta_b, y) = occ(\theta_b, z) = \emptyset$. The reader is left 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 [10, 11, 12, 13]. 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 3.1 $[\alpha_X]$ 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 in [11, 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 3.1 α_X is surjective. ▌

However, α_X is not injective, or in other words, 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 3.1 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'	$\cup S'$	$X \setminus (\cup S')$
$\emptyset$	$\emptyset$	$\{x, y, z\}$	$\{G_3\}$	$\{z\}$	$\{x, y\}$
$\{G_1\}$	$\{x\}$	$\{y, z\}$	$\{G_1, G_3\}$	$\{x, z\}$	$\{y\}$
$\{G_2\}$	$\{y, z\}$	$\{x\}$	$\{G_2, G_3\}$	$\{y, z\}$	$\{x\}$
$\{G_1, G_2\}$	$\{x, y, z\}$	$\emptyset$	$\{G_1, G_2, G_3\}$	$\{x, y, z\}$	$\emptyset$

Therefore $\alpha_X(S) = \mathit{model}_X{}^{-1}(\{\emptyset, \{x\}, \{y\}, \{x, y\}, \{y, z\}, \{x, y, z\}\}) = (y \leftarrow z)$. 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 [12, 13]. $\mathcal{C}$ 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 3.2 $[rel]$ The map $rel : \wp(X) \times \mathit{Share}_X \to \mathit{Share}_X$ is defined by:

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

Definition 3.3 The map $\mathcal{C} : \mathit{Share}_X \to \mathit{Def}_X$ is defined by $\mathcal{C}(S) = \wedge F$ where

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

Example 3.2 Consider again Example. 3.1. The set of $Y \subseteq X \setminus \{x\}$ such that $rel(\{x\}, S) \subseteq rel(Y, S)$ is $\{\{x\}, \{x, y\}, \{x, z\}, \{x, y, z\}\}$. Likewise, set of $Y \subseteq X \setminus \{y\}$ such that $rel(\{y\}, S) \subseteq rel(Y, S)$ is $\{\{y\}, \{z\}, \{x, z\}, \{x, y\}, \{y, z\}, \{x, y, z\}\}$. Finally, the set of $Y \subseteq X \setminus \{z\}$ such that $rel(\{z\}, S) \subseteq rel(Y, S)$ 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 in [11], albeit for slightly different definitions. Modifying their proof to our definitions is straightforward.

Proposition 3.2 $\mathcal{C} = \alpha_X$. ▮

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 [19], we obtain a useful lemma about these equivalence classes.

Definition 3.4 Let $S \in \mathit{Share}_X$. Then the closure under union $S^\star$ of S is defined by: $S^\star = \{\cup S' \mid S' \subseteq S\} \setminus \{\emptyset\}$. ▮

Lemma 3.3 $S^\star \equiv S$ for all $S \in \mathit{Share}_X$. $S_1 \equiv S_2$ iff $S_1^\star = S_2^\star$ for all $S_1, S_2 \in \mathit{Share}_X$. ▮

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 3.5 $[\sqsubseteq, \sqcup, \sqcap]$

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

Proposition 3.4 $\langle \mathit{Share}_X/\!\equiv, \sqsubseteq, \sqcap, \sqcup \rangle$ 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 efficiently with $Share_X/\equiv$ via the isomorphism. It is not straightforward 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 [12, 13] that with the *Share* representation, *join* essentially reduces to set union.

Proposition 4.1 $[S_1]_\equiv \sqcup [S_2]_\equiv = [S_1 \cup S_2]_\equiv$ ∎

Example 4.1 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 [19]. We follow [13] and denote binary union as $\otimes$.

Definition 4.1 [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 [19]. 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 $f \wedge (\wedge Y_1 \leftarrow \wedge Y_2)$ respectively.

Definition 4.2 The two maps $\mathit{iff} : \wp(X) \times \wp(X) \times Share_X \to Share_X$ and $\mathit{if} : \wp(X) \times \wp(X) \times Share_X \to Share_X$ are defined by:

$$\mathit{iff}(Y_1, Y_2, S) = (S \setminus (S_1 \cup S_2)) \cup (S_1 \otimes S_2) \qquad \mathit{if}(Y_1, Y_2, S) = (S \setminus S_1) \cup (S_1 \otimes S_2)$$

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 [19] 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 [3]. In the following we state the safety of *iff* and *if*.

Proposition 4.2 $\alpha_X(\mathit{iff}(Y_1, Y_2, S)) = \alpha_X(S) \wedge (\wedge Y_1 \leftrightarrow \wedge Y_2)$ ∎

Corollary 4.3 $\alpha_X(\mathit{if}(Y_1, Y_2, S)) = \alpha_X(S) \wedge (\wedge Y_1 \leftarrow \wedge Y_2)$ ∎

Even though $\mathit{if}(Y_1, Y_2, S)$ can be simulated with $\mathit{iff}(Y_1', Y_2, S)$ where $Y_1' = Y_1 \cup Y_2$, $rel(Y_1, S)$ is cheaper to compute than $rel(Y_1', S)$ and this is one reason why $\mathit{if}(Y_1, Y_2, S)$ is more efficient than $\mathit{iff}(Y_1', Y_2, S)$. 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 [18]. For completeness, we state its correctness as a proposition.

Definition 4.3 [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 4.4 If $Y \subseteq X$ then $\exists(X \setminus Y).\alpha_X([S]_\equiv) = \alpha_Y([\overline{\exists} Y.S]_\equiv)$. ∎

Finally, Theorem 4.5 shows how *meet* can be computed with a sequence of *iff* calculations.

Theorem 4.5 $[S_1]_\equiv \sqcap [S_2]_\equiv = [\overline{\exists} X.S'_{n+1}]_\equiv$ where $X = \{x_1, \ldots, x_n\}$ and

$$\begin{array}{rcl} S'_1 & = & \rho(S_1) \cup S_2 \\ S'_{j+1} & = & \mathit{iff}(\{\rho(x_j)\}, \{x_j\}, S'_j) \quad \text{for } j \in \{1, \ldots, n\}, \end{array}$$

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 4.2 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 widening

Since formulae are represented as equivalence classes, the fixpoint stability check, $f = f'$, amounts to checking whether $[S]_\equiv = [S']_\equiv$ which, in turn, reduces to deciding whether or not $\alpha_X(S) = \alpha_X(S')$ where $\alpha_X(S) = f$ and $\alpha_X(S') = f'$. Since Share_X is finite and $S = S'$ implies $f = f'$, in theory at least, termination can be obtained by merely checking $S = S'$. Such an analysis may perform more iterations than is strictly necessary. This approach turns out to be impractical because the efficiency of an analysis depends on (1) the number of iterations and (2) the cost of each iteration which, in turn, relates to the size of the abstractions. We thus introduce a compression operator, c, that (1) helps stability detection and (2) keeps the abstractions small.

Definition 5.1 $c : \mathit{Share}_X \to \mathit{Share}_X$ is defined by: $c(S) = \cap\{S' \mid S' \equiv S\}$. ∎

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

Proposition 5.1 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 3.3. 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 large numbers of redundant variable sets as is illustrated in the following example. Because of this compression is always applied after projection.

Example 5.1 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 redundant set $\{x, y\}$, whereas $c(\overline{\exists}\{x, y\}.S) = \{\{x\}, \{y\}\}$. ▮

Apart from reducing the size of abstractions, it is also worthwhile to avoid generating large abstractions, which 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 5.2

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

where $S_1 = rel(Y_1, S)$, $S_2 = rel(Y_2, S)$, $|S| = n$, $|S_1| = n_1$ and $|S_2| = n_2$. ▮

This widening ensures that at each stage of the analysis the size of an abstraction is either smaller than k or smaller the abstraction at the previous program point. A widened *meet* can be obtained by replacing each $\mathit{iff}(\{\rho(x_j)\}, \{x_j\}, S'_j)$ of Theorem 4.5 by $\mathit{iff}_k(\{\rho(x_j)\}, \{x_j\}, S'_j)$.

6 Experimental work

To investigate whether a quadratic widened *meet* is 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 [8] and eager evaluation [24] to perform goal-dependent bottom-up evaluation. Induced magic is a refinement of the magic set transformation [5], avoiding much of the re-computation that arises because of the repetition of literals in the bodies of magicked clauses [8]. It also avoids the overhead of applying the magic set transformation. Eager evaluation [24] 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 reevaluated. 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 the overhead of computing SCCs.

Table 1 summarises our experimental results for applying Def in definiteness analysis [4] to some of the largest Prolog and CLP($\mathcal{R}$) benchmark programs that we could find on the WWW. The first ten programs are CLP($\mathcal{R}$) sources whereas the others are Prolog sources. To assess the precision of the Def analyses (widened to various k), we have implemented a standard Pos analysis following the technique of [9]. We are not arguing that the times for Pos are optimal, though we have invested a lot of effort fine-tuning the Pos analysis. The *abs* column gives 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 for an unwidened Def analysis with compression; Def analyses with compression widened to $k = 0$, $k = 64$ and $k = 256$; and the Pos analysis. Times are in seconds and > 60 indicates that the fixpoint calculation timed out after one minute. The timings were obtained on a one-processor 167 MHz SPARC Ultra with 256 MByte of memory running Solaris 2.5.1. The analyses were coded in SICStus 3.5. 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 bracketed numbers in *precision* columns for the *Def* analyses indicate the number of occasions when widening has been applied in an *iff* or an *if* computation.

The experimental results indicate that *Def* has better scaling behaviour than the *Pos* implementation of [9]: there are no programs for which *Pos* terminates within a minute and *Def* does not. What is surprising is that the *Def* analysis compares favourably to the *Pos* analyses for speed even for programs, like `aircraft`, that are very well suited to the *Pos* analysis of [9], that is, the programs contain very large numbers of ground arguments. This is because a program which contains a large number of ground arguments is likely to have small sharing abstractions. Widening seems to be crucial for scalability as is illustrated by `csg` and `reducer`. (This also suggests that widening is necessary in the [3] pair-sharing quotient of *Share* though this is not discussed.) The precision of *Def* with $k = 0$ is another surprising result since precision is lost with respect to *Pos* only for `rings-on-pegs` and `qplan`! This is because the size of the initial (call) abstractions for each predicate is typically greater than zero. A threshold of $k = 64$ seems to be a simple pragmatic way of preserving precision while achieving scalability. To summarise, the experimental results are very encouraging and despite the simplicity of the interpreter, our widened *Def* analysis appears to be fast, precise and scalable and, of course, can be implemented easily in Prolog.

7 Related work

In [12] it was first pointed out that *Share* expresses the groundness dependencies of *Def*. Quotienting was introduced in [13] as a systematic way of obtaining the reference domain of [12]. Like [3], we do not fully adhere to the quotienting terminology and methodology of [12] but rather follow the standard convention [14] of inducing an equivalence relation ($\equiv$) from an abstraction map (α_X). Also, Lemma 6.2 of [13] can be interpreted as a way of computing the *meet* in *Def* with the classic abstract unification of [19]. We take this further and show how the *meet* can be computed without exponential time closure operations.

In [3] it is pointed out that *Share* includes redundant sharing information with respect to pair-sharing. This work is related to ours in that their construction also quotients *Share*. In fact, our domain may be viewed as a further quotient of the pair-sharing domain and the *meet* and *join* operators in [3] are similar to ours. 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.

In [1] various normal forms of Boolean functions and the relative precision of *Pos* and *Def* are investigated. C-based implementations of a bottom-up analyser based on each representation are described. For the representations 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 *meet* operation, however, may have exponential cost [1, Section 4.2]. 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.

China [2] is a C-based analyser for the Pattern(*Pos*) domain [2, 23] that uses ROBDD's for the *Pos* component. It is a bottom-up analyser that uses magic to capture separate call patterns for each call site. China takes around 20 seconds to analyse `trs` on a 233 MHz Pentium II PC. For the other benchmarks that we have in common, our dependency analysis seems to be similar in speed to Pattern(*Pos*) though is is hard to make accurate comparisons because China runs on a faster machine but uses a more fine-grained framework.

file	*abs*	*fixpoint*					*precision*				
		Def	Def_0	Def_{64}	Def_{256}	Pos	Def	Def_0	Def_{64}	Def_{256}	Pos
air	0.02	3.88	0.11	0.21	0.41	1.47	9	(22) 9	(6) 9	(2) 9	9
bridge	0.14	0.02	0.01	0.02	0.03	0.16	24	(1) 24	24	24	24
circuit	0.03	0.88	0.11	0.15	0.16	> 60	3	(18) 3	(8) 3	(3) 3	
csg	0.11	> 60	0.12	0.24	0.93	> 60		(210) 12	(179) 12	(161) 12	
dnf	0.04	0.03	0.02	0.02	0.02	0.01	8	8	8	8	8
laplace	0.02	10.98	1.41	1.24	1.33	> 60	0	(22) 0	(16) 0	(14) 0	
mortgage	0.01	10.86	0.01	0.04	0.13	0.01	6	(30) 6	(15) 6	(12) 6	6
neural	0.08	0.25	0.19	0.25	0.25	0.17	118	(16)118	118	118	118
option	0.02	7.93	0.12	0.25	0.48	> 60	42	(101) 42	(75) 42	(61) 42	
rings-on-pegs	0.03	0.13	0.11	0.13	0.13	0.73	11	(50) **5**	11	11	11
aircraft	0.57	0.51	0.56	0.49	0.57	0.20	609	609	609	609	609
ann	0.15	0.83	0.50	0.54	0.55	3.04	74	(13) 74	(1) 74	74	74
asm	0.17	0.12	0.15	0.14	0.14	0.11	87	(1) 87	87	87	87
boyer	0.08	0.05	0.04	0.05	0.04	0.04	18	18	18	18	18
browse	0.03	0.04	0.03	0.03	0.02	0.03	42	42	42	42	42
bryant	0.08	0.64	0.32	0.56	0.69	7.48	89	(19) 89	(13) 89	89	89
conman	0.07	0.01	0.01	0.01	0.00	0.00	6	6	6	6	6
chat_80	1.31	5.73	5.48	5.63	5.69	> 60	839	(89)839	(3)839	839	
chat_parser	0.32	1.83	1.67	1.81	1.91	> 60	505	(29)505	505	505	
crypt_wamcc	0.03	0.08	0.09	0.08	0.08	0.03	31	31	31	31	31
cs_r	0.08	0.05	0.04	0.02	0.05	0.03	149	149	149	149	149
dbqas	0.10	0.05	0.04	0.02	0.04	0.24	43	43	43	43	43
dialog	0.03	0.03	0.03	0.04	0.03	0.02	70	70	70	70	70
disj_r	0.02	0.04	0.07	0.05	0.06	0.01	97	97	97	97	97
essln	0.42	0.86	0.90	0.90	0.90	0.51	121	121	121	121	121
flatten	0.04	0.44	0.22	0.32	0.41	3.43	27	(17) 27	(4) 27	27	27
ga	0.08	0.05	0.04	0.04	0.03	0.01	138	138	138	138	138
gabriel	0.02	0.09	0.08	0.08	0.08	0.08	37	(8) 37	37	37	37
ili	0.19	2.20	0.60	1.33	1.76	48.62	4	(153) 4	(43) 4	(3) 4	4
kalah	0.07	0.06	0.03	0.05	0.04	0.03	199	199	199	199	199
ime_v2-2-1	0.06	0.23	0.23	0.24	0.25	0.73	101	101	101	101	101
life	0.03	0.03	0.03	0.03	0.03	0.00	58	58	58	58	58
lnprolog	0.13	0.15	0.14	0.08	0.15	0.10	172	172	172	172	172
mastermind	0.03	0.04	0.06	0.07	0.06	0.05	43	43	43	43	43
nand	0.15	0.37	0.32	0.37	0.33	0.09	286	286	286	286	286
nandc	0.03	0.04	0.04	0.03	0.05	0.08	37	37	37	37	37
nbody	0.10	3.67	0.25	0.28	0.32	4.49	109	(21)109	(6)109	(3)109	109
neural	0.09	0.08	0.08	0.09	0.09	0.08	123	123	123	123	123
parser_dcg	0.12	0.25	0.27	0.23	0.33	0.68	41	41	41	41	41
peep	0.19	0.16	0.20	0.17	0.15	0.68	10	(4) 10	10	10	10
peval	0.20	0.91	0.39	0.50	0.86	42.13	28	(10) 28	(26) 28	28	28
poly10	0.03	0.00	0.00	0.00	0.00	0.00	0	0	0	0	0
press	0.09	5.18	0.47	0.74	1.10	14.44	53	(79) 53	(20) 53	(5) 53	53
qplan	0.10	0.84	0.64	0.75	0.82	> 60	49	(21) **48**	(2) 49	49	
read	0.13	0.34	0.37	0.32	0.34	12.96	79	(7) 79	79	79	79
reducer	0.09	> 60	0.30	0.51	0.59	> 60		(14) 43	(25) 43	(16) 43	
robot	0.05	0.01	0.00	0.01	0.01	0.00	37	37	37	37	37
ronp	0.02	0.11	0.09	0.09	0.10	0.32	10	10	10	10	10
rubik	0.25	0.47	0.48	0.39	0.43	0.87	203	203	203	203	203
scc1	0.03	0.07	0.07	0.08	0.10	0.06	89	89	89	89	89
sdda	0.07	0.15	0.14	0.16	0.18	53.31	14	(11) 14	14	14	14
semi	0.04	0.78	0.74	0.71	0.75	10.77	26	(6) 26	26	26	26
sim	0.23	1.58	1.30	1.40	1.56	> 60	100	(10)100	(3)100	100	
simple_analyzer	0.10	0.95	0.87	0.97	0.94	> 60	89	(17) 89	89	89	
sim_v5-2	0.28	0.16	0.17	0.16	0.17	0.48	101	(1)101	101	101	101
strips	0.22	0.06	0.05	0.04	0.06	0.02	140	140	140	140	140
tictactoe	0.07	0.03	0.03	0.03	0.02	0.01	49	49	49	49	49
trs	0.17	1.39	1.30	1.46	1.42	> 60	13	(91) 13	13	13	
tsp	0.03	0.09	0.10	0.09	0.10	0.04	122	122	122	122	122
unify	0.06	0.34	0.38	0.34	0.35	> 60	70	(7) 70	70	70	

Table 1: Experimental evaluation

In [18], Def functions are represented 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 of [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.

A widening for Pos, $WPos$, is proposed in [6] that amounts to a sub-domain of Def that cannot propagate dependencies of the form $y \leftarrow (y \wedge z)$, but only bidirectional dependencies of the form $(x \leftrightarrow y)$. The main finding of [6] is that $WPos$ looses only a small amount of precision for Prolog programs. We have found, however, that $WPos$ can loose up to 50% of the ground arguments of some constraint programs. This, presumably, is because constraints, like $x = y + z$, give rise to formulae of the form $(x \leftarrow (y \wedge z)) \wedge (y \leftarrow (x \wedge z)) \wedge (z \leftarrow (x \wedge y))$. Our Def analysis solves this precision problem.

8 Future work

Future work will focus on strictness analysis [20] and investigate whether Mon_X can be efficiently represented and manipulated with *Share*. On the framework side, we will also investigate how call-graph information can be exploited, for example, to determine how immediate fixpoints [22] can be used to avoid Kleene iteration (for directly recursive predicates at least).

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. Our work reinforces the observation that in abstract interpretation there is a strong synergy between theory and practice.

Acknowledgements

We thank Mike Codish and Andy Heaton for discussions on widening that motivated the study, Mark Wallace and Samir Genaim for useful comments, and Roberto Bagnara for help with the benchmarks. This work was funded partly by the EPSRC Grant GR/K97635.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency analysis. *Science of Computer Programming*, 1998. to appear.

[2] R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages.* PhD thesis, Dipartimento di Informatica, March 1997.

[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, Long version submitted to *Theoretical Computer Science.*

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

[5] C. Beeri and R. Ramakrishnan. On the power of magic. *J. Logic Programming*, 10(3&4):255–300, 1991.

[6] F. Benoy, M. Codish, A. Heaton, and A. King. Widening *Pos* for Efficient and Scalable Groundness Analysis. Technical Report 515, University of Kent at Canterbury, 1997. http://www.cs.ukc.uk/pubs/1997/515/index.local.

[7] 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.

[8] M. Codish. Efficient Goal Directed Bottom-up Evaluation of Logic Programs. Technical report, Ben-Gurion University of the Negev, 1998. ftp://ftp.cs.bgu.ac.il/pub/people/mcodish/fastmagic.ps.gz.

[9] 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.

[10] A. Cortesi and G. Filé. Comparison and Design of Abstract Domains for Sharing Analysis. In *Italian Conference on Logic Programming*, pages 251–265. Mediterranean Press, 1993.

[11] A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in Abstract Interpretation. *ACM TOPLAS*, 19(1):7–47, January 1997.

[12] 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.

[13] A. Cortesi, G. Filé, and W. Winsborough. The Quotient of an Abstract Interpretation for Comparing Static Analyses. In *Joint Conference on Declarative Programming (GULP-PRODE '94)*, pages 372–397, 1994. An extended version will appear in *Theoretical Computer Science*.

[14] 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.

[15] P. Dart. On Derived Dependencies and Connected Databases. *J. Logic Programming*, 11(1&2):163–188, 1991.

[16] 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.

[17] C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung.* PhD thesis, Universität des Saarlandes, 1997. http://www.cs.uni-sb.de/RW/users/fecht/.

[18] 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.

[19] D. Jacobs and A. Langen. Static Analysis of Logic Programs. *J. Logic Programming*, 13(2 and 3):154–314, 1992.

[20] L. Mauborgne. Abstract interpretation using TDGs. In *Static Analysis Symposium*, pages 363–379. Springer-Verlag, 1994. LNCS 864.

[21] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *J. Logic Programming*, 13:315–347, 1992.

[22] H. Søndergaard. Immediate fixpoints and their use in groundness analysis. In *Foundations of Software Technology and Theoretical Computer Science*, pages 359–370. Springer-Verlag, 1996. LNCS 1180.

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

[24] J. Wunderwald. Memoing Evaluation by Source-to-Source Transformation. In *Logic Program Synthesis and Transformation*, pages 17–32. Springer, 1995. LNCS 1048.

A Proofs (for referees only)

Proof (Proposition 2.1) By definition of $\models$, $f \models g$ iff $model_X(f) \subseteq model_X(g)$ and $f = g$ iff $model_X(f) = model_X(g)$. Suppose $S_1 \subset S_2 \subset \ldots S_k$ is a chain such that

- $S_1 = \{X\}$,
- for all $i \in \{1, \ldots, k-1\}$, $S_{i+1} = S_i \cup \{Y_{i+1}\}$ for some $Y_{i+1} \in \wp(X)$, where Y_{i+1} has minimal cardinality such that $Y_{i+1} \notin S_i$, and
- $S_k = \wp(X)$.

S_1 is the smallest set S such that $model_X^{-1}(S) \in Def_X$. By construction, each S_i is closed under intersection and thus $model_X^{-1}(S_i) \in Def_X$. Furthermore, since $|S_{i+1}| = |S_i| + 1$ for all $i \in \{1, \ldots, k-1\}$, and since $Y_{i+1} \in \wp(X)$ for all $i \in \{1, \ldots, k-1\}$ and $|\wp(X)| = 2^n$, it follows that $k = 2^n$ and the chain is maximal. ∎

Proof (Lemma 3.1) Consider an arbitrary $f \in Def_X$. We show that for $S = \{X \setminus M \mid M \in model_X(f)\}$, we have $\alpha_X(S) = f$.

$$
\begin{array}{ll}
\alpha_X(S) = & \text{Defn. 3.1} \\
\quad model_X^{-1}(\{X \setminus \cup S' \mid S' \subseteq \{X \setminus M \mid M \in model_X(f)\}\}) = & (\textit{trivial}) \\
\quad model_X^{-1}(\{X \setminus (X \setminus \cap \mathcal{M}) \mid \mathcal{M} \subseteq model_X(f)\}) = & (\textit{trivial}) \\
\quad model_X^{-1}(\{\cap \mathcal{M} \mid \mathcal{M} \subseteq model_X(f)\}) = & (*) \\
\quad model_X^{-1}(model_X(f)) = & (\textit{trivial}) \\
\quad f. &
\end{array}
$$

All steps just use simple set transformations. However, the crucial step $(*)$ exploits that by definition, $model_X(f)$ is closed under intersection. ∎

Proof (Lemma 3.3) For the first statement, let $S \in Share_X$. We must show that $\alpha_X(S) = \alpha_X(S^\star)$. By Def. 3.1,

$$
\begin{array}{rcl}
\alpha_X(S) & = & model_X^{-1}(\{X\} \cup \{X \setminus G \mid G \in S^\star\}) \\
\alpha_X(S^\star) & = & model_X^{-1}(\{X\} \cup \{X \setminus G \mid G \in (S^\star)^\star\}).
\end{array}
$$

However, the closure under union is idempotent so that $(S^\star)^\star = S^\star$ and hence the result.

For the second statement observe that

$$
\begin{array}{l}
S_1 \equiv S_2 \iff \\
\quad \alpha_X(S_1) = \alpha_X(S_2) \iff \\
\quad \{X \setminus G \mid G \in S_1^\star\} = \{X \setminus G \mid G \in S_2^\star\} \iff \\
\quad \{G \in S_1^\star\} = \{G \in S_2^\star\} \iff \\
\quad S_1^\star = S_2^\star.
\end{array}
$$

∎

Proof (Proposition 3.4) Since $Share_X/\equiv$ is finite, it remains to be shown that $\langle Share_X/\equiv, \sqsubseteq, \sqcap, \sqcup\rangle$ is a lattice. Let $S_1, S_2 \in Share_X$. We only show that $[S_1]_\equiv \sqcup [S_2]_\equiv$ is the *join*. Showing that $[S_1]_\equiv \sqcap [S_2]_\equiv$ is the *meet* is similar.

Let $i \in \{1, 2\}$. Since $\alpha_X([S_i]_\equiv) \models \alpha_X([S_1]_\equiv) \dot\vee \alpha_X([S_2]_\equiv)$ it follows that $[S_i]_\equiv = {\alpha_X}^{-1}(\alpha_X([S_i]_\equiv)) \sqsubseteq {\alpha_X}^{-1}(\alpha_X([S_1]_\equiv) \dot\vee \alpha_X([S_2]_\equiv)) = [S_1]_\equiv \sqcup [S_2]_\equiv$. Therefore $[S_i]_\equiv \sqsubseteq [S_1]_\equiv \sqcup [S_2]_\equiv$.

Let $S \in Share_X$ such that $[S_1]_\equiv \sqsubseteq [S]_\equiv$ and $[S_2]_\equiv \sqsubseteq [S]_\equiv$. Since $\alpha_X([S_1]_\equiv) \models \alpha_X([S]_\equiv)$ and $\alpha_X([S_2]_\equiv) \models \alpha_X([S]_\equiv)$, it follows that $\alpha_X([S_1]_\equiv) \dot\vee \alpha_X([S_2]_\equiv) \models \alpha_X([S]_\equiv)$ and hence $[S_1]_\equiv \sqcup [S_2]_\equiv = {\alpha_X}^{-1}(\alpha_X([S_1]_\equiv) \dot\vee \alpha_X([S_2]_\equiv)) \sqsubseteq [S]_\equiv$. Therefore $[S_1]_\equiv \sqcup [S_2]_\equiv \sqsubseteq [S]_\equiv$. ∎

Proof (Proposition 4.1) Since by Definition 3.5, $[S_1]_\equiv \sqcup [S_2]_\equiv = {\alpha_X}^{-1}(\alpha_X([S_1]_\equiv)\dot{\vee}\alpha_X([S_2]_\equiv))$, we must show ${\alpha_X}^{-1}(\alpha_X([S_1]_\equiv)\dot{\vee}\alpha_X([S_2]_\equiv)) = [S_1 \cup S_2]_\equiv$. Since α_X is bijective, this is equivalent to showing that $\alpha_X([S_1]_\equiv)\dot{\vee}\alpha_X([S_2]_\equiv) = \alpha_X([S_1 \cup S_2]_\equiv)$. By the definition of lifting α_X to equivalence classes, this is in turn equivalent to showing that $\alpha_X(S_1)\dot{\vee}\alpha_X(S_2) = \alpha_X(S_1 \cup S_2)$. Thus we now establish that $\alpha_X(S_1 \cup S_2) \models \alpha_X(S_1)\dot{\vee}\alpha_X(S_2)$ and that $\alpha_X(S_1)\dot{\vee}\alpha_X(S_2) \models \alpha_X(S_1 \cup S_2)$.

- We first show $\alpha_X(S_1 \cup S_2) \models \alpha_X(S_1)\dot{\vee}\alpha_X(S_2)$. Suppose $f \in \mathit{Def}_X$ such that $\alpha_X(S_1) \models f$ and $\alpha_X(S_2) \models f$. Let $M \in \mathit{model}_X(\alpha_X(S_1 \cup S_2))$. By Definition 3.1, $M = X \setminus ((\cup {S_1}') \cup (\cup {S_2}'))$, where ${S_1}' \subseteq S_1$ and ${S_2}' \subseteq S_2$. Likewise, by Definition 3.1, $X \setminus (\cup {S_1}') \in \mathit{model}_X(\alpha_X(S_1))$ and $X \setminus (\cup {S_2}') \in \mathit{model}_X(\alpha_X(S_2))$ and therefore $X \setminus (\cup {S_1}') \in \mathit{model}_X(f)$ and $X \setminus (\cup {S_2}') \in \mathit{model}_X(f)$. Since $f \in \mathit{Def}_X$, it follows that $M = (X \setminus (\cup {S_1}')) \cap (X \setminus (\cup {S_2}')) \in \mathit{model}_X(f)$. Thus $\alpha_X(S_1 \cup S_2) \models f$. Therefore, since the choice of f was arbitrary, $\alpha_X(S_1 \cup S_2) \models \wedge\{f \in \mathit{Def}_X \mid \alpha_X(S_1) \models f \wedge \alpha_X(S_2) \models f\}$ and hence, by the definition of $\dot{\vee}$, $\alpha_X(S_1 \cup S_2) \models \alpha_X(S_1)\dot{\vee}\alpha_X(S_2)$.

- We now show $\alpha_X(S_1)\dot{\vee}\alpha_X(S_2) \models \alpha_X(S_1 \cup S_2)$. Suppose $M \in \mathit{model}_X(\alpha_X(S_1)\dot{\vee}\alpha_X(S_2))$. By definition of $\dot{\vee}$, $M \in \mathit{model}_X(f)$ for all $f \in \mathit{Def}_X$ such that $\alpha_X(S_1) \models f$ and $\alpha_X(S_2) \models f$. By Def. 3.1, $\mathit{model}_X(\alpha_X(S_i)) \subseteq \mathit{model}_X(\alpha_X(S_1 \cup S_2))$ and thus $\alpha_X(S_i) \models \alpha_X(S_1 \cup S_2)$ for $i \in \{1, 2\}$. Furthermore by the well-definedness of α_X, $\alpha_X(S_1 \cup S_2) \in \mathit{Def}_X$. Hence it follows that $M \in \mathit{model}_X(\alpha_X(S_1 \cup S_2))$, and since the choice of M was arbitrary, $\alpha_X(S_1)\dot{\vee}\alpha_X(S_2) \models \alpha_X(S_1 \cup S_2)$. ▮

Proof (Proposition 4.2) Let $S \in \mathit{Share}_X$, $Y_1, Y_2 \subseteq X$ and $S_1 = \mathit{rel}(Y_1, S)$, $S_2 = \mathit{rel}(Y_2, S)$. Let $f = \alpha_X(S)$ and $f' = \alpha_X(\mathit{iff}(Y_1, Y_2, S))$.

- We show $f \wedge (\wedge Y_1 \leftrightarrow \wedge Y_2) \models f'$. Suppose $M \in \mathit{model}_X(f)$ and $M \in \mathit{model}_X(\wedge Y_1 \leftrightarrow \wedge Y_2)$. We must show that $M \in \mathit{model}_X(f')$. By Definition 3.1, $M = X \setminus (\cup S')$ for some $S' \subseteq S$. Since $M \in \mathit{model}_X(\wedge Y_1 \leftrightarrow \wedge Y_2)$, there are two main cases to consider.
  - Suppose $Y_1 \cup Y_2 \subseteq M$. Then, since $M = X \setminus (\cup S')$, it follows that $(Y_1 \cup Y_2) \cap (\cup S') = \emptyset$. Thus $S' \subseteq S \setminus (S_1 \cup S_2)$ and hence $M \in \mathit{model}_X(f')$ by Definitions 3.1 and 4.2.
  - Suppose $Y_1 \not\subseteq M$ and $Y_2 \not\subseteq M$.
    * Suppose $(\cup S') = (\cup S_a) \cup (\cup S_b)$ where $S_a \subseteq S \setminus (S_1 \cup S_2)$ and $\emptyset \subset S_b \subseteq S_1$. Since $Y_2 \not\subseteq M$, $Y_2 \cap (\cup S') \neq \emptyset$ and thus there exists $G_2 \in S_b$ such that $G_2 \cap Y_2 = \emptyset$. Thus $(\cup S') = (\cup S_a) \cup (\cup (S_b \otimes \{G_2\}))$ where $S_b \otimes \{G_2\} \subseteq S_1 \otimes S_2$. Since $S_a \subseteq S \setminus (S_1 \cup S_2)$ it follows that $M \in \mathit{model}_X(f')$.
    * Suppose $(\cup S') = (\cup S_a) \cup (\cup S_c)$ where $S_a \subseteq S \setminus (S_1 \cup S_2)$ and $\emptyset \subset S_c \subseteq S_2$. Similar to the previous case.
    * Suppose $(\cup S') = (\cup S_a) \cup (\cup S_b) \cup (\cup S_c)$ where $S_a \subseteq S \setminus (S_1 \cup S_2)$, $\emptyset \subset S_b \subseteq S_1$ and $\emptyset \subset S_c \subseteq S_2$. Let $G_1 \in S_b$ and $G_2 \in S_c$. Thus $(\cup S') = (\cup S_a) \cup (\cup(\{G_1\} \otimes S_c)) \cup (\cup(S_b \otimes \{G_2\}))$ where $\{G_1\} \otimes S_c \in S_1 \otimes S_2$ and $S_b \otimes \{G_2\} \in S_1 \otimes S_2$. Since $S_a \subseteq S \setminus (S_1 \cup S_2)$ it follows that $M \in \mathit{model}_X(f')$.

- We show $f' \models f \wedge (\wedge Y_1 \leftrightarrow \wedge Y_2)$. Let $M \in \mathit{model}_X(f')$. By Definitions 3.1 and 4.2, $M = X \setminus (\cup S')$ for some $S' \subseteq \mathit{iff}(Y_1, Y_2, S)$.
  - We must show $M \in \mathit{model}_X(\wedge Y_1 \leftrightarrow \wedge Y_2)$.
    * Suppose $S' \subseteq (S \setminus (S_1 \cup S_2))$. Then $S' \cap Y_1 = \emptyset$ and $S' \cap Y_2 = \emptyset$. Therefore $Y_1 \cup Y_2 \subseteq M$ and thus $M \in \mathit{model}_X(\wedge Y_1 \leftrightarrow \wedge Y_2)$.

* Suppose $S' \cap (S_1 \otimes S_2) \neq \emptyset$. Since $G_1 \cap Y_1 \neq \emptyset$ for all $G_1 \in S_1$ and $G_2 \cap Y_2 \neq \emptyset$ for all $G_2 \in S_2$, it follows that $G \cap Y_1 \neq \emptyset$ and $G \cap Y_2 \neq \emptyset$ for all $G \in (S_1 \otimes S_2)$. Therefore there is a $G \in S'$ such that $G \cap Y_1 \neq \emptyset$ and $G \cap Y_2 \neq \emptyset$, and so there are variables $y_1 \in Y_1$ and $y_2 \in Y_2$ such that $y_1, y_2 \in \cup S'$ and hence $y_1, y_2 \notin M$. Therefore $M \in model_X(\wedge Y_1 \leftrightarrow \wedge Y_2)$.

- We must show that $M \in model_X(f)$. There are S_a, S_b such that $S' = S_a \cup S_b$ and $S_a \subseteq S \setminus (S_1 \cup S_2)$ and $S_b \subseteq (S_1 \otimes S_2)$. Now $S_b = \{G_1 \dots G_n\}$ where $G_i = G_{i,1} \cup G_{i,2}$, and $G_{i,1} \in S_1$, $G_{i,2} \in S_2$ for all $i \in \{1 \dots n\}$. Since $\cup S' = (\cup S_a) \cup (\cup_{i=1}^{n} G_{i,1}) \cup (\cup_{i=1}^{n} G_{i,2})$ and $S_a, \{G_{i,1} \mid i = 1 \dots n\}, \{G_{i,2} \mid i = 1 \dots n\} \subseteq S$, it follows that $M \in model_X(f)$. ▮

Proof (Corollary 4.3) Let $Y_1, Y_2 \subseteq X$, $S_1 = rel(Y_1, S)$ and $S_2 = rel(Y_2, S)$. Since $(\wedge Y_1 \leftarrow \wedge Y_2) = (\wedge(Y_1 \cup Y_2) \leftrightarrow \wedge Y_2)$, by Proposition 4.2 it follows that $\alpha_X(S) \wedge (\wedge(Y_1 \cup Y_2) \leftrightarrow \wedge Y_2) = (S \setminus ((S_1 \cup S_2) \cup S_2)) \cup (S_1 \cup S_2) \otimes S_2 = (S \setminus S_1) \cup (S_1 \otimes S_2)$. ▮

Proof (Proposition 4.4) Let $Y \subseteq X$. We show $\exists(X \setminus Y).\alpha_X(S) = \alpha_Y(\overline{\exists} Y.S)$. Since $var(\alpha_Y(\overline{\exists} Y.S) \subseteq Y$, it follows that

$$
\begin{aligned}
& model_X(\alpha_Y(\overline{\exists} Y.S)) = \\
& \quad \{M_1 \cup M_2 \mid M_1 \in model_Y(\alpha_Y(\overline{\exists} Y.S)) \wedge M_2 \subseteq X \setminus Y\} = \\
& \quad \{(Y \setminus (\cup S')) \cup M_2 \mid S' \subseteq (\overline{\exists} Y.S) \wedge M_2 \subseteq X \setminus Y\} = \\
& \quad \{(((X \setminus (\cup S')) \cap Y) \cup M_2 \mid S' \subseteq S \wedge M_2 \subseteq X \setminus Y\} = \\
& \quad \{(M_1 \cap Y) \cup M_2 \mid M_1 \in model_X(\alpha_X(S)) \wedge M_2 \subseteq X \setminus Y\} = \\
& \quad model_X(\exists(X \setminus Y).\alpha_X(S)).
\end{aligned}
$$

▮

Proof (Theorem 4.5) Note that

$$
\begin{aligned}
& model_{\rho(X)\cup X}(\rho(\alpha_X(S_1)) \wedge \alpha_X(S_2)) = \\
& \quad model_{\rho(X)\cup X}(\rho(\alpha_X(S_1))) \cap model_{\rho(X)\cup X}(\alpha_X(S_2)) = \\
& \quad \{(\rho(X) \cup X) \setminus ((\cup S_1{}') \cup (\cup S_2{}')) \mid S_1{}' \subseteq \rho(S_1) \wedge S_2{}' \subseteq S_2\} = \\
& \quad model_{\rho(X)\cup X}(\alpha_{\rho(X)\cup X}(\rho(S_1) \cup S_2)).
\end{aligned}
$$

Thus $\rho(\alpha_X(S_1)) \wedge \alpha_X(S_2) = \alpha_{\rho(X)\cup X}(\rho(S_1) \cup S_2)$.

Now $\alpha_X(S_1) \wedge \alpha_X(S_2) = \exists \rho(X).(\rho(\alpha_X(S_1)) \wedge \alpha_X(S_2) \wedge (\wedge_{x \in X} \rho(x) \leftrightarrow x)) = \exists \rho(X).\alpha_X(S'_{n+1})$ because of proposition 4.2. By proposition 4.4 it follows that $\alpha_X(S_1) \wedge \alpha_X(S_2) = \alpha_X(\overline{\exists} X.S'_{n+1})$ as required. ▮

Proof (Proposition 5.1) We must show that $c(S) \subseteq S_n$ and $S_n \subseteq c(S)$.

- By construction of S_n it follows that $S_n{}^\star = S^\star$. Therefore by Lemma 3.3, $S_n \equiv S$. Thus it follows that $c(S) \subseteq S_n$.
- For the other direction, suppose $S' \equiv S$ so that, by Lemma 3.3, $S^\star = S'^\star$. We first show that each element $G \in S^\star$ is a union of elements of $S \cap S'$.

  The proof is by induction on $|G|$. If $|G| = 1$, then $G \in S \cap S'$ and the results holds. Suppose $|G| > 1$ and $G \notin S \cap S'$.

  - First case: $G \in S$. Since $G \notin S \cap S'$ it follows that $G \notin S'$. But since $G \in S'^\star$ it follows that $G = G_1 \cup G_2$ where $G_1, G_2 \in S'^\star$ and $|G_1|, |G_2| < |G|$. Since $S^\star = S'^\star$, it follows that $G_1, G_2 \in S^\star$.
  - Second case: $G \notin S$. Since $G \in S^\star$ it follows that $G = G_1 \cup G_2$ where $G_1, G_2 \in S^\star$ and $|G_1|, |G_2| < |G|$.

Thus in both cases $G_1, G_2 \in S^\star$ and $|G_1|, |G_2| < |G|$. Then, by the inductive hypothesis, each G_1, G_2 is the union of elements of $S' \cap S$. Hence G is the union of elements of $S' \cap S$.

Suppose that $G \in S_n$. Then, since $S_n \subseteq S$, $G \in S$ and thus $G \in S^\star$. Hence, by the previous paragraph, $G = G_1 \cup \ldots \cup G_m$ and each $G_i \in S \cap S'$. Since this means in particular that $G_i \in S$, it follows by the construction of S_n that $G_i \in {S_n}^\star$. Now if $|G_i| < |G|$ for all $i \in \{1, \ldots, m\}$ then by construction of S_n, $G \notin S_n$ which is a contradiction. Thus there exists $i \in \{1, \ldots, m\}$ such that $|G_i| \geq |G|$. Hence $G_i = G$ and $G \in S \cap S'$.

Since this is true for all $S' \equiv S$, we have $G \in c(S)$. ∎

B Benchmark suite (for referees only)

`air.clpr` – calculate excess air needed for a chemical reaction from CLP($\mathcal{R}$) version 2.0 release; `aircraft.pl` – air traffic verification system from the University of Huddersfield; `ann.pl` – simplified clause annotator by Manual Hermenegildo, R. Warren, and M. Muthukumar; `asm.pl` – SB-Prolog assembler; `boyer.pl` – an extract from a Boyer-Moore theorem prover by Evan Tick after Lisp by Bob Boyer; `bridge.clpr` – bridge simulation program; `browse.pl` – build and query a database by Tep Dobry and Herve' Touati; `bryant.pl` – manipulate ordered binary decision diagrams by Peter Schachte; `chat_80.pl` – natural language query of a geographical database by Fernando Pereira and David Warren; `chat_parser.pl` – parse a set of English sentences by Fernando Pereira and David Warren; `circuit.clpr` – simple circuit simulator; `conman.pl` – an expert system shell with uncertainty by Michael Covington, Donald Nute, and Andre Vellino; `crypt_wamcc.pl` – Cryptomultiplication by Peter Van Roy; `cs_r.pl` – program for cutting stock; `csg.clpr` – intersection of solids in CSG representation; `dbqas.pl` – a query-answering system by Tu Van Le; `dcg.pl` – Kumon's grammar ICOT/GIGALIPS benchmark by Evan Tick; `dialog.pl` – a simple-system dialogue program by Tu Van Le; `disj_r.pl` – disjunctive scheduling program; `dnf.clpr` – convert a propositional formula into disjunctive normal form from CLP($\mathcal{R}$) version 1.1 release; `essln.pl` – an expert system shell by Tu Van Le; `flatten.pl` – source transformation to remove disjunctions; `ga.pl` – simple genetic algorithm from the University of Uppsala; `gabriel.pl` – an alpha-beta search problem; `ili.pl` – an intuitionistic logic interpreter by Seif Haridi and Translated to Quintus Prolog by Evan Tick; `ime_v2-2-1.pl` – Gaussian elimination solver by unknown author; `kalah.pl` – kalah program by Leon Sterling and Ehud Shapiro; `laplace.clpr` – solve the Dirichlet problem for Laplace's equation from CLP($\mathcal{R}$) version 1.1 release; `life.pl` – "Game of Life" program by Jamie Andrews and William Clocksin; `lnprolog.pl` – meta-interpreter for LnProlog by Tu Van Le; `mastermind.pl` – mastermind game by Herbert Koenig and Thomas Hoppe; `mortgage.clpr` – standard mortgage relationship from CLP($\mathcal{R}$) version 1.1 release; `nand.pl` – a logic synthesis program based on heuristic search by Bruce Holmer; `nandc.pl` – interactive game of noughts and crosses by David Green; `nbody.pl` – n-body problem for star clusters by Evan Tick; `neural.clpr` – neural network program; `neural.pl` – simulator of a network; `option.clpr` – algebraic combinations of options transactions from CLP($\mathcal{R}$) version 1.1 release; `parser_dcg.pl` – dcg parser an imperative and a functional language by Roberto Bagnara; `peep.pl` – peephole optimiser from SB-Prolog; `peval.pl` – interpreter for flow chart language of Jones, Gomard and Sestoft; `poly10.pl` – raise a polynomial to the 10th power by Ralph Haygood, Rick McGeer and Richard Gabriel; `press.pl` – Rubik's cube program by D Merritt; `qplan.pl` – query plan of Chat-80 by David HD Warren; `reducer.pl` – a graph reducer based for T-combinators by Peter Van Roy; `rings-on-pegs.clpr` – place a ring on each peg without breaking the string; `robot.pl` – instruct a robot to perform some house-work using best-cost search by Michael Covington; `ronp.pl` – place a ring on each peg of the board combinatorial puzzle; `rubik.pl` – Rubik's cube puzzle; `scc1.pl` – compute the strongly connected components of a graph by Stephan Diehl; `sdda.pl` – an analyser that capture aliasing; `semi.pl` – semigroup generator by Mats Carlsson, Evan Tick and Ross Overbeek; `sim.pl` – a simulator for OR-parallel Prolog by Kish Shen; `sim_v5-2.pl` – an algebraic simplifier by Lin Zhuhail of South Bank University; `simple_analyzer.pl` – dataflow analyser for quicksort by Peter Van Roy as a part of the Aquarius project; `strips.pl` – strips planning from Artificial Intelligence Techniques in Prolog by Yoav Shoham; `tictactoe.pl` – tic-tac-toe by alpha-beta pruning by Arvind Kumar Bansal; `trs.pl` – theorem prover for Gentzen's sequent calculus LK by K. Sakai; `tsp.pl` – near-optimal solution to the Travelling Salesman Problem by the University of Uppsala; `unify.pl` – a compiler code generator for unification from the Aquarius Prolog compiler.