Abstract Compilation for Sharing Analysis

Gianluca Amato[1] and Fausto Spoto[2]

[1] Dipartimento di Informatica, Corso Italia, 40, I-56100 Pisa, Italy
`amato@di.unipi.it`
Ph.: +39-050887248   Fax: +39-050887226
[2] IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
`spoto@irisa.fr`

Abstract. An abstract domain for non pair-sharing and freeness analysis of logic programs has been recently developed by using the *automatic* technique of linear refinement. W.r.t. previously available domains, it can be used for abstract compilation, which allows a modular and goal-independent analysis of logic programs. In this paper, we describe our implementation of an analyser which uses that domain. Sometimes, we have sacrificed precision for efficiency. We evaluate it over a set of benchmarks and we compare the results with those obtained through a goal-dependent analysis. Not surprisingly, our goal-independent analysis is slower. However, it is almost always as precise as the goal-dependent one. To the best of our knowledge, this is the first goal-independent implementation of sharing analysis based on abstract interpretation, as well as the first implementation of a linearly refined domain.

Keywords: *Abstract interpretation, domain theory, linear refinement, linear logic, logic programming.*

1 Introduction

Pair-sharing analysis [1, 16] determines those pairs of variables which, in a given program point, can be bound to two terms which share some variable. It is a particular case of set-sharing analysis [12]. In set-sharing analysis, indeed, sets of variables are considered, and not just pairs. It is useful for avoiding occur-check [16] and for automatic program parallelisation [12, 15]. As stressed in [1], pair-sharing information is actually needed in program analysis and transformation, set-sharing information being redundant w.r.t. pair-sharing information.

Freeness analysis [4, 15] determines those variables which are always bound to a variable in a given program point. It is useful for optimising unification, for goal reordering and for avoiding type checking. It is well known that performing sharing and freeness analysis together improves the precision of both [12, 15].

When the fixpoint computation is based on a compositional definition, the $(i + 1)$-th iteration can re-use any intermediate results already computed during the i-th iteration that are known not to change across iterations. Such results are usually the denotations of some program parts which do not contain recursive procedure calls. Therefore, these parts can be replaced by their denotation and the fixpoint computed on this modified (partially *compiled*) program. This technique is traditionally known as *abstract compilation* [5, 11], since it is an application of abstract interpretation [9] where a program is iteratively compiled to its abstract denotation. This leads, in general, to a more efficient computation of the abstract fixpoint. Moreover, modular analysis is allowed. This means that procedures or libraries can be analysed separately, and their analyses can *then* be combined to obtain the analysis of a large program.

Linear refinement [10] is a technique for systematically improving abstract domains for program analysis. It has been defined as a slight generalisation of Cousot's reduced power operation [8]. Given a basic abstract domain representing just the property of interest and a concrete operation $\boxtimes$ (in the case of logic programming, unification) the new refined domain is constructed, and allows us to define an abstract operation which is more precise than that of the basic domain. This is achieved by enriching the basic domain with linear logic implications $i \multimap o$ representing the propagation of the abstract property of interest through the concrete operator $\boxtimes$. Namely, if the abstract property i holds for the input of the operator $\boxtimes$ then the abstract property o holds for its output. In this way, the development of new domains becomes almost automatic and we can define the denotation for a procedure as a function from abstract properties of the input to abstract properties of the output. Thus, static analysis can be applied even if the source code of some procedures is not known (which can be the result of some copyright policy), provided that its abstract denotation is available.

In [13] an abstract domain for non pair-sharing and freeness analysis has been developed through linear refinement. It is more precise than the traditional domain of [12] and correct abstract operators have been explicitly defined. However, no experimental results are provided. Since the abstract operations are not optimal, the usefulness of the domain was left unclear.

Our contribution is the implementation of the domain of [13], which is the first implementation of a goal-independent sharing and freeness analysis of logic programs based on abstract interpretation, as well as of a linearly refined domain. Note that the traditional domains for groundness did exist before it became clear they could be obtained through refinement. Beyond the implementation, this paper contains a piece of theory about *reduction rules*, necessary for an efficient analysis, and which can be tuned in such a way to avoid any loss of precision.

1.1 Related Works

Almost all works about sharing analysis are not amenable to abstract compilation and have been developed without any automatic technique like linear refinement. To the best of our knowledge, only [4, 6, 7] provide abstract domains for sharing analysis which can be used for abstract compilation. The domain in [4] models sharing, freeness and groundness. It is not based on abstract interpretation. Its authors claim that its precision is no more than that of the

Sharing $\times$ *Free* domain of [12]. An implementation exists. The domain in [6] is isomorphic to the *Sharing* domain of [12]. The domain in [7] is isomorphic to the domain *Pos* for groundness analysis. Since in both cases the authors provided an abstract unification algorithm over the abstract domain only (in contrast with an abstract unification algorithm between an element of the abstract domain and a concrete substitution, like in [12]), abstract compilation is allowed in both cases. However, the domains in [6] and [7] induce abstraction maps which are too coarse for practical applications. Namely, those maps cannot distinguish between concrete substitutions like $\{x = y\}$ and $\{x = \mathtt{f}(y, y)\}$ (as implied by Equation (7) of [6] and Observation 4.1 of [7]). However, those substitutions must be distinguished in order to provide decent precision, as shown in Example 8 of [13]. Other sources of imprecision are shown by Examples 4 and 5 of [13]. Some of those problems can be solved by adding *freeness* and *linearity* to the abstract domain, but we are not aware of any implementation of the domains in [6] and [7] coupled with freeness and linearity.

2 Preliminaries

We denote by $\wp(S)$ the powerset of a set S, by $\#S$ its cardinality and by $\wp_f(S)$ the set of all finite subsets of S.

Given a set of variables V and a set of function symbols Σ with associated arity, containing at least a symbol of arity 0, we define $\mathsf{terms}(\Sigma, V)$ as the minimal set of terms built from V and Σ, i.e., $V \subseteq \mathsf{terms}(\Sigma, V)$ and if $t_1, \ldots, t_n \in \mathsf{terms}(\Sigma, V)$ and $\mathtt{f}^n \in \Sigma$, then $\mathtt{f}(t_1, \ldots, t_n) \in \mathsf{terms}(\Sigma, V)$. We denote by $\mathsf{vars}(t)$ the set of variables which occur in a term t. When $\mathsf{vars}(t) = \emptyset$ we say that t is ground. If x is a variable, $V \cup x$ means $V \cup \{x\}$ and $V \setminus x$ means $V \setminus \{x\}$. The set of idempotent substitutions θ ($\mathsf{dom}(\theta) \cup \mathsf{rng}(\theta) \subseteq V$ and $\mathsf{dom}(\theta) \cap \mathsf{rng}(\theta) = \emptyset$) is denoted by Θ_V.

Let $\mathcal{V}$ be an infinite set of variables and $V \in \wp_f(\mathcal{V})$. We define the set $C_V = \wp_f\{t^1 = t^2 \mid t^1, t^2 \in \mathsf{terms}(\Sigma, V)\}$ of *Herbrand constraints*. Let $\mathcal{W}$ be an infinite set of variables disjoint from $\mathcal{V}$. For each $V \in \wp_f(\mathcal{V})$, we have the set of *existential Herbrand constraints*

$$H_V = \left\{ \exists_W c \;\middle|\; \begin{array}{l} W \in \wp_f(\mathcal{W}),\ c \in C_{V \cup W} \text{ and there exists} \\ \theta \in \Theta_{V \cup W} \text{ s.t. } \mathsf{rng}(\theta) \subseteq V \text{ and } c\theta \text{ holds} \end{array} \right\} .$$

Here, $\mathcal{V}$ are called the *program variables* and $\mathcal{W}$ the *existential variables*, which are a formalisation of the unnamed variables of Prolog. The condition about the existence of θ such that $c\theta$ holds, means that we consider satisfiable constraints only.

Four operations, called *conjunction*, *restriction*, *expansion* and *renaming*, are defined over H_V.

Definition 1. *We define* $\star^{H_V} : H_V \times H_V \mapsto H_V$, $\mathsf{restrict}_n^{H_V} : H_V \mapsto H_{V \setminus n}$ *with* $n \in V$, $\mathsf{expand}_x^{H_V} : H_V \mapsto H_{V \cup x}$ *with* $x \notin V$, *and* $\mathsf{rename}_{x \to n}^{H_V} : H_V \mapsto H_{(V \setminus x) \cup n}$, *with $x \in V$ and $n \notin V$ as*[1]

$$
\begin{aligned}
(\exists_{W_1} c_1) \star^{H_V} (\exists_{W_2} c_2) &= \begin{cases} \exists_{W_1 \cup W_2} \mathsf{mgu}(c_1 \cup c_2) & \textit{if } \mathsf{mgu}(c_1 \cup c_2) \textit{ exists,} \\ \textit{undefined} & \textit{otherwise} \end{cases} \\
\mathsf{restrict}_n^{H_V}(\exists_W c) &= \exists_{W \cup N} c[N/n] \qquad \textit{with } N \in \mathcal{W} \setminus W, \\
\mathsf{expand}_x^{H_V}(\exists_W c) &= \exists_W c \ , \\
\mathsf{rename}_{x \to n}^{H_V}(\exists_W c) &= \exists_W (c[n/x]) \ .
\end{aligned}
$$

Note that in the definition of $\star^{H_V}$ we put the constraint in normal form through the Martelli and Montanari unification algorithm [14]. The other operations are closed on the set of existential Herbrand constraints in normal form.

Our concrete domain is the collecting version [9] of H_V, i.e., the lattice $\langle \wp(H_V), \cap, \cup, H_V, \emptyset \rangle$. The operations on H_V are point-wise extended to $\wp(H_V)$. The new operation $\cup^{\wp(H_V)}(S_1, S_2) = S_1 \cup S_2$ is defined. It is used to merge the results of different branches of execution.

3 Non Pair-Sharing and Freeness Analysis

We briefly recall the definition of the abstract domain for non pair-sharing and freeness described in [13]. Its abstract elements are sets of arrows.

Definition 2. *Given $V \in \wp_f(\mathcal{V})$, we denote by V_2 the set of* unordered *pairs of elements of V. We define $ShF_V = \wp(V \cup V_2)$ as the domain used to express the freeness of variables and the non-sharing of pairs of variables. We define $Abs_V = \wp(ShF_V \times (V \cup V_2))$. We write the elements of V_2 as (v_1, v_2), with $\{v_1, v_2\} \subseteq V$ and a pair $(\{l_1, \ldots, l_n\}, r) \in Abs_V$ as $l_1 \cdots l_n \Rightarrow r$, for $n \geq 0$. The* dimension *$\mathsf{dim}(s)$ of $s \in ShF$ is its cardinality. If $A \in Abs_V$, $\mathsf{dim}(A) = \sum_{l \Rightarrow r \in A} \mathsf{dim}(l)$.*

For instance, the object $x(y, z) \in ShF_V$ means that x is free and that y and z do not share any variable. The arrow $l_1 \cdots l_n \Rightarrow r$ represents the set of existential Herbrand constraints which, when unified with a constraint satisfying $l_1 \cdots l_n$ (i.e., whose freeness and non pair-sharing properties are consistent with those expressed by $l_1 \cdots l_n$), give a result satisfying r.

We can approximate the operations of Definition 1. *Entailment* and *tautological* arrows are defined in [13]. Roughly speaking, $l \in ShF_V$ entails $l' \in ShF_V$ when every existential Herbrand constraint satisfying l satisfies l', i.e., its freeness and non pair-sharing properties, when consistent with those expressed by l are also consistent with those expressed by l'. For instance, we have that $x(y, x)$ entails (y, x) and x, and that (x, x) entails (x, y) (since (x, x) means that x is ground). A tautological arrow is an arrow which is satisfied by every constraint.

[1] In the definition of $\star^{H_V}$ we assume $W_1 \cap W_2 = \emptyset$, since a constraint $\exists_W c$ is equivalent to $\exists_{W'} c[W'/W]$, with W' made of fresh variables. Similarly, different choices of N in $\mathsf{restrict}_n^{H_V}$ lead to equivalent constraints.

Definition 3. *Let $A_1, A_2 \in Abs_V$. Let T be made of tautological arrows*[2]. *Then*

$$A_1 \star^{Abs_V} A_2 = \left\{ l_1 \cdots l_n \Rightarrow r \;\middle|\; \begin{array}{l} r_1 \cdots r_n \Rightarrow r \in A_2 \cup T,\ l_i \Rightarrow r'_i \in A_1 \cup T \\ \text{ and } r'_i \text{ entails } r_i \text{ for } i = 1, \ldots, n, \text{ or} \\ r_1 \cdots r_n \Rightarrow r \in A_1 \cup T,\ l_i \Rightarrow r'_i \in A_2 \cup T \\ \text{ and } r'_i \text{ entails } r_i \text{ for } i = 1, \ldots, n \end{array} \right\} .$$

In our implementation, in Definition 3 we use $T = \{(v, v) \Rightarrow (v, v) \mid v \in V\}$.

Example 1. In Definition 3, let $T = \{\}$, $V = \{x, y, z\}$ and

$$\begin{aligned} A_1 &= \{xy \Rightarrow x, (x, y)(x, z) \Rightarrow (x, y), (x, x) \Rightarrow (z, z)\} , \\ A_2 &= \{x(x, y) \Rightarrow x,\ \Rightarrow (y, y)\} . \end{aligned}$$

Then

$$A_1 \star^{Abs_V} A_2 = \{xy(x, y)(x, z) \Rightarrow x,\ \Rightarrow (y, y)\} .$$

Note that we do not obtain the arrow $\{(x, x) \Rightarrow (z, z)\}$, since the set T is empty. But this arrow must hold for $A_1 \star^{Abs_V} A_2$, since it holds for A_1 and groundness dependencies cannot be lost. By using $T \supseteq \{(v, v) \Rightarrow (v, v) \mid v \in V\}$ we would include that arrow in the conjunction.

Definition 4. *Let $V \in \wp_f(\mathcal{V})$ and $n \in V$. Let $X = \{n\} \cup \{(v, n) \mid v \in V\}$ and $A, A_1, A_2 \in Abs_V$. We define*

$$\mathsf{restrict}_n^{Abs_V}(A) = \left\{ l \setminus X \Rightarrow r \;\middle|\; \begin{array}{l} l \Rightarrow r \in A,\ (n, n) \notin l, \\ r \not\equiv n \text{ and } r \not\equiv (n, v) \text{ for every } v \in V \end{array} \right\} .$$

If $x \in V$, $n \notin V$ and $A \in Abs_V$, we define

$$\mathsf{rename}_{x \to n}^{Abs_V}(A) = A[n/x] .$$

Finally, we define

$$\cup^{Abs_V}(A_1, A_2) = \{l_1 l_2 \Rightarrow r \mid l_1 \Rightarrow r \in A_1,\ l_2 \Rightarrow r \in A_2\} .$$

Example 2. Let $V = \{x, y, z\}$ and

$$A = \{xy \Rightarrow x, xy \Rightarrow y, (x, x) \Rightarrow (y, z), (y, z)(x, y)(x, z) \Rightarrow (y, z)\} .$$

Then

$$\mathsf{restrict}_x^{Abs_V}(A) = \{y \Rightarrow y, (y, z) \Rightarrow (y, z)\} .$$

For the expansion, we use a distinguished variable $?_1 \in V$ which stands for all the variables which do not occur in the Herbrand constraints. In order to compute $\mathsf{expand}_x^{Abs_V}(A)$, we substitute x for $?_1$ in A. Since non pair-sharing is a property of pairs of variables, we wish to know how this new x behaves in conjunction with the distinguished variable itself. Thus we use two distinguished variables $?_1$ and $?_2$. By using the abstraction map (Definition 6) with $\{?_1, ?_2\} \subseteq V$, we introduce these variables in the constraints.

[2] The larger T is, the more precise $\star^{Abs_V}$ is.

Definition 5. *Let $V \in \wp_f(\mathcal{V})$ and $x \in \mathcal{V} \setminus V$. Let $\{?_1, ?_2\} \subseteq V$ and $A \in Abs_V$. We define*

$$\mathsf{expand}_x^{Abs_V}(A) = A \cup \{l[x/?_1][?_1/?_2] \Rightarrow r[x/?_1][?_1/?_2] \mid l \Rightarrow r \in A\} \ .$$

Example 3. Let $V = \{x, y, z\}$ and

$$A = \{?_1(x, ?_1) \Rightarrow ?_1, (?_1, x)(?_1, z)(?_1, ?_2) \Rightarrow (?_1, ?_2), xyz \Rightarrow z\} \ .$$

Given $n \in \mathcal{V} \setminus V$ we have

$$\mathsf{expand}_n^{Abs_V}(A) = \left\{ \begin{array}{l} xyz \Rightarrow z, n(x, n) \Rightarrow n, (n, x)(n, z)(n, ?_1) \Rightarrow (n, ?_1), \\ ?_1(x, ?_1) \Rightarrow ?_1, (?_1, x)(?_1, z)(?_1, ?_2) \Rightarrow (?_1, ?_2) \end{array} \right\} \ .$$

We show here how to compute an approximation of the abstraction map. It considers separately the information of groundness, non pair-sharing and freeness contained in a binding. A substitution is then abstracted by combining through $\star^{Abs_V}$ the abstraction of every binding of which it is composed.

Definition 6. *Let $V \in \wp_f(\mathcal{V})$, $v \in V$ and $t \in \mathsf{terms}(\Sigma, V)$. We define*

$$\alpha^V(v = t) = \alpha^V_{gr}(v = t) \cup \alpha^V_{nsh}(v = t) \cup \alpha^V_{fr}(v = t) \ ,$$

where α^V_{gr}, α^V_{nsh} and α^V_{free} are defined below. We write $t(v_1, \ldots, v_n)$ for t if $\mathsf{vars}(t) = \{v_1, \ldots, v_n\}$. If $n = 0$ then $t(v_1, \ldots, v_n)$ is ground. Variables with different names are different variables.

$$\alpha^V_{gr}(v = t(v_1, \ldots, v_n)) = \bigcup_{v' \in V} \left\{ \begin{array}{l} (v_1, v_1) \cdots (v_n, v_n) \Rightarrow (v, v'), \\ (v, v) \Rightarrow (v_1, v'), \ldots, (v, v) \Rightarrow (v_n, v') \end{array} \right\}$$

$$\begin{aligned}
\alpha^V_{nsh}(x = t) &= \cup_{\{v,v'\} \subseteq V} \alpha^{(v,v')}_{nsh}(x = t) \\
\alpha^{(v,v')}_{nsh}(v = t(v', v_1, \ldots, v_n)) &= \{\} = \alpha^{(v,v')}_{nsh}(v = t') \quad t' \textit{ ground} \\
\alpha^{(v,v')}_{nsh}(x = t(v, v', v_1, \ldots, v_n)) &= \{x(v, v') \Rightarrow (v, v')\} \\
\alpha^{(v,v')}_{nsh}(v = t(v_1, \ldots, v_n)) &= \{(v', v)(v', v_1) \cdots (v', v_n) \Rightarrow (v', v)\} \\
&\qquad t(v_1, \ldots, v_n) \textit{ non ground} \\
\alpha^{(v,v')}_{nsh}(x = t(v, v_1, \ldots, v_n)) &= \left\{ \begin{array}{l} (v', v)(v', x)(v', v_1) \cdots (v', v_n) \Rightarrow (v', v), \\ (v', v)(v', x)x \Rightarrow (v', v) \end{array} \right\} \\
\alpha^{(v,v')}_{nsh}(x = t) &= \{(v, v') \Rightarrow (v, v')\} \quad t \textit{ ground} \\
\alpha^{(v,v')}_{nsh}(x = t(v_1, \ldots, v_n)) &= \left\{ \begin{array}{l} (v, v')(v, x)(v, v_1) \cdots (v, v_n) \Rightarrow (v, v'), \\ (v, v')(v', x)(v', v_1) \cdots (v', v_n) \Rightarrow (v, v'), \\ (v, v')(v, x)(v', x)x \Rightarrow (v, v') \end{array} \right\} \\
&\qquad t(v_1, \ldots, v_n) \textit{ non ground}
\end{aligned}$$

$$\begin{aligned}
\alpha^V_{fr}(x = t) &= \cup_{v\in V} \alpha^v_{fr}(x = t)\\
\alpha^v_{fr}(v = x) &= \{vx \Rightarrow v\} = \alpha^v_{fr}(x = t(v, v_1, \ldots, v_n))\\
\alpha^v_{fr}(v = t(v_1, \ldots, v_n)) &= \{\} \quad t(v_1, \ldots, v_n) \notin \mathcal{V}\\
\alpha^v_{fr}(x = y) &= \{v(x, v)(v, y) \Rightarrow v, vxy \Rightarrow v\}\\
\alpha^v_{fr}(x = t(v_1, \ldots, v_n)) &= \{vx(x, v) \Rightarrow v, v(x, v)(v_1, v) \cdots (v_n, v) \Rightarrow v\}\\
&\qquad t(v_1, \ldots, v_n) \notin \mathcal{V}\ .
\end{aligned}$$

We can compute the abstract information contained in an abstract constraint, i.e., the set of variables which are free and the set of pairs of variables which do not share in any existential Herbrand constraint belonging to the concretisation of the abstract constraint.

Definition 7. *Given* $V \in \wp_f(\mathcal{V})$, *we define* $\mathsf{free}_V : Abs_V \mapsto \wp(V)$ *and* $\mathsf{nsh}_V : Abs_V \mapsto V_2$ *as*

$$\begin{aligned}
\mathsf{free}_V(A) &= \{v \in V \mid l \Rightarrow v \in A \textit{ and } v'(v', v') \not\subseteq l \textit{ for any } v' \in V\}\ ,\\
\mathsf{nsh}_V(A) &= \{(v_1, v_2) \in V_2 \mid l \Rightarrow (v_1, v_2) \in A \textit{ and } (v, v) \notin l \textit{ for any } v \in V\}\ .
\end{aligned}$$

4 Implementation

We describe now our prototypical implementation[3] of the domain of Section 3. We did not aim at efficiency, though much care has been taken to avoid the explosion of the computational cost of the abstract conjunction operator (Definition 3).

Constraints are manipulated by C procedures, while the normalisation, abstraction and fixpoint computation phases (see below) are written in Prolog. The choice of C as implementation language for the constraints is a consequence of efficiency considerations. Indeed, constraints are represented by arrays of bitmaps. Every basic *token* of information, i.e., the freeness of a variable or the non-sharing of a pair of variables, is associated with a bit position. Elements of ShF_V (Definition 2) are then implemented as strings of bits.

4.1 Normalisation, Abstraction and Fixpoint Computation

We describe the three phases of our analysis, normalisation, abstraction and fixpoint calculation of the abstract s-semantics for computed answers [3] (a call-pattern or resultant semantics could be used here), by using the traditional `member/2` program as a running example:

```
member(X,[X|Xs]).
member(X,[Y|Ys]):-member(X,Ys).
```

[3] Downloadable at `http://www.di.unipi.it/∼amato/papers/flops01.tgz`.

Normalisation transforms a program in such a way that procedure calls are made only in their most general form, i.e., with variables `v(0)`, `v(1)`, and so on, as arguments. Moreover, the structure of the program (disjunctions, conjunctions, expansions and similar) is made apparent, which simplifies the subsequent fixpoint iteration. For instance, the normalisation of `member/2` is

```
member(2):-
   rename(v(16), v(0), rename(v(17), v(1), or(
      restrict(v(18), bi_eq(v(17), [v(16)|v(18)])),
      restrict(v(20), and(
          expand(v(16), restrict(v(19), bi_eq(v(17), [v(19)|v(20)]))),
          expand(v(17), rename(v(0), v(16), rename(v(1), v(20),
              call(member(2)))))
      ))
   )))
```

The program has been compiled in a code which contains calls to the abstract operations of the domain, as well as built-in's, like `bi_eq`, which unifies two terms, and procedure calls, like `call(member(2))`, where 2 is the arity of the procedure. This preliminary transformation has the advantage of keeping the set of variables used for the analysis of a given program point as small as possible. For instance, if a variable is not used in a constraint, then it will be added (through `expand`) *after* the analysis (i.e., the abstraction) of the constraint. See the case of `v(16)` and `v(17)` in the normalisation of `member/2` above. If a variable is not used after a conjunction, then it can be removed (through `restrict`). See the case of `v(20)` in the normalisation of `member/2` above. This reduces the complexity of the analysis, since the abstract operations have computational complexities which are proportional to the dimension of the elements of the abstract domain, and this dimension is in its turn proportional to the number of variables used.

Note that the normalisation phase above is not abstract compilation. Instead, abstract compilation substitutes the built-in's with their abstract behaviour, i.e., a constraint of the abstract domain. In the case of `bi_eq` it applies the abstraction map of Definition 6. Moreover, it applies partial evaluation when the operands of an abstract operation are known (i.e., if they do not contain any procedure call), by substituting the operation with its result. In our case, we obtain

```
member(2):-
   rename(v(16), v(0), rename(v(17), v(1), or(
       abs(177072),
       restrict(v(20), and(
           abs(203050),
           expand(v(17), rename(v(0), v(16), rename(v(1), v(20),
               call(member(2)))))
       ))
   )))
```

An element of Abs_V is written as abs(N), N being the pointer in memory where the constraint is stored. Indeed, as we have said, constraints are manipulated by C procedures.

The calculation of the abstract fixpoint is just an iterated depth-first evaluation of the abstract program, by using the operations of Section 3. The call graph of the program is used. Namely, if a procedure p is called by a procedure q but not vice versa (even through intermediate procedures), the denotation of p is computed first, and that of q later. In the case of `member/2`, this machinery is of no help.

Even for a procedure as simple as `member/2`, our abstract analyser would not reach the abstract fixpoint in a reasonable time, since the computational cost of the analysis explodes. We show now how to obtain a reasonable performance.

4.2 Reduction Rules and Widening

Since several elements of Abs_V may have the same concretisation (γ_{Rep} is defined in the proofs appendix) and the computational complexity of the abstract operators of Section 3 depends on the dimension of their operands, we wish to use the elements of Abs_V of smallest dimension. This is the goal of *reduction rules.*

Definition 8. *A* reduction rule *is a family of maps* $\{\rho_V\}_{V\in\wp_f(\mathcal{V})}$ *such that, for every* $V \in \wp_f(\mathcal{V})$*,*

i) $\rho_V : Abs_V \mapsto Abs_V$,
ii) $\mathsf{dim}(\rho_V(A)) \le \mathsf{dim}(A)$ *and*
iii) $\gamma_{Rep}(\rho_V(A)) = \gamma_{Rep}(A)$ *for every* $A \in Abs_V$*.*

The last two conditions say that a reduction rule reduces the dimension of a constraint without losing any information. Though it can be shown that not all the operations of Section 3 are monotonic w.r.t. dim, our evaluation (Section 5) suggests that reduction rules are useful in practice. If we apply a reduction rule after every abstract operator, Definition 8 guarantees that we obtain a correct result. The following condition entails that we do not lose any precision, which was not obvious, since the abstract operators are not optimal.

Proposition 1. *For* $V \in \wp_f(\mathcal{V})$ *and* $A_1, A_2 \in Abs_V$*, let* $A_1 \preceq A_2$ *if and only if for every* $l_2 \Rightarrow r \in A_2$ *there is* $l_1 \Rightarrow r \in A_1$ *with*

$$l_1 \subseteq l_2 \cup \{(v,v') \mid v,v' \in V,\ v \neq v' \text{ and } (v,v) \in l_2\} \ .$$

Every reduction rule ρ *which is reductive w.r.t.* $\preceq$ *(i.e.,* $\rho(A) \preceq A$ *for every* $A \in Abs_V$*) does not introduce any loss of precision.*

We show now two examples of reduction rules which are reductive w.r.t. $\preceq$.

Proposition 2. *Let* $\rho^1 = \{\rho^1_V\}_{V\in\wp_f(\mathcal{V})}$*, where*

$$\rho^1_V(A) = \left\{ l \Rightarrow r \in A \;\middle|\; \begin{array}{l} \text{there is no } l' \Rightarrow r \in A \text{ s.t.} \\ l' \subset l \cup \{(v,v') \mid v,v' \in V,\ v \neq v' \text{ and } (v,v) \in l\} \end{array} \right\}$$

for any $V \in \wp_f(\mathcal{V})$ *and* $A \in Abs_V$*. Then* ρ^1 *is a reduction rule and is reductive w.r.t.* $\preceq$*. Moreover, it is possible to prove that* $\rho^1_V(A) = \bigcap\{X \subseteq A \mid X \preceq A\}$*, i.e.,* $\rho^1(A)$ *is the smallest set of* A *to precede* A *w.r.t.* $\preceq$*.*

Example 4. Let $V = \{v, x, y, z\}$ and

$$A = \{x(x,y)(x,z)(x,v) \Rightarrow (x,v), x(v,v) \Rightarrow (x,v), xy \Rightarrow y, x(x,v) \Rightarrow (x,v)\} .$$

Then

$$\rho^1_V(A) = \{xy \Rightarrow y, x(x,v) \Rightarrow (x,v)\} .$$

Proposition 3. *Let $\rho^2 = \{\rho^2_V\}_{V \in \wp_f(\mathcal{V})}$, where*

$$\rho^2_V(A) = \{l \setminus (\{(v,v') \mid v, v' \in V,\ v \neq v' \text{ and } (v,v) \in l\} \Rightarrow r) \mid l \Rightarrow r \in A\}$$

for any $V \in \wp_f(\mathcal{V})$ and $A \in Abs_V$. Then ρ^2 is a reduction rule and is reductive w.r.t. $\preceq$.

Example 5. Let $V = \{v, x, y, z\}$ and

$$A = \{x(x,y)(y,y)(y,z)(x,z) \Rightarrow (x,z), xy \Rightarrow y, y(y,z)(z,z)(v,v) \Rightarrow y\} .$$

Then

$$\rho^2_V(A) = \{x(y,y)(x,z) \Rightarrow (x,z), xy \Rightarrow y, y(z,z)(v,v) \Rightarrow y\} .$$

The efficiency of the analysis can be obviously improved by removing some arrows from the elements of Abs_V, possibly introducing some imprecision, like with every widening operation [9]. In our implementation we use syntactical equality for the entailment test of Definition 3, which means that some arrows allowed by the theory are not generated by the implementation.

4.3 The Result of the Analysis

By using the techniques of Subsection 4.2, our analyser computes the following denotation for `member/2`:

```
?1,(a1,?1),(a0,?1),(a0,a1)=>?1
?1,a1,(a1,?1)=>?1
(?1,?2),(a1,?1),(a0,?1),(a0,a1)=>(?1,?2)
(?1,?1)=>(?1,?1)
(a1,?1),(a0,?1),(a0,a1)=>(a1,?1)
(a1,a1)=>(a1,a1)
a1,a0,(a0,a1)=>a0
(a1,?1),(a0,?1),(a0,a1)=>(a0,?1)
(a1,a1)=>(a0,?1)
(a0,a0)=>(a0,a0)
(a1,a1)=>(a0,a0)
```

The variables `a0` and `a1` are the two argument positions of the procedure. Beyond simple groundness dependencies, note that the analyser concludes that $a1, a0, (a0, a1) \Rightarrow a0$. Indeed, if `member/2` is called with two different free variables, then the freeness of the first variable cannot be lost. Note that the simple freeness of `a1` and `a2` is not judged enough to this purpose. Indeed, a call like `member(X,X)` binds `X` to the term `[X|_]` if the occur-check is not applied. If the occur-check is applied, the freeness of `a1` and `a2` would be enough.

Benchmark	Bytes	Prepr.	Fix.	Conj.	Others	Shell
ackermann.pl	139	0.06	0.10	35.0%	7.2%	57.8%
append.pl	62	0.03	0.52	77.5%	6.9%	16.5%
eliza.pl	1400	0.42	2.20	70.1%	6.0%	23.9%
hanoi.pl	199	0.09	2.91	88.9%	4.3%	5.7%
heapify.pl	508	0.16	106.67	99.5%	0.0%	0.5%
map_coloring.pl	419	0.08	0.49	57.6%	12.9%	29.5%
queens.pl	735	0.24	1.01	52.7%	14.9%	33.3%
quicksort.pl	431	0.18	18.45	97.0%	1.0%	2.0%
openlist.pl	159	0.76	0.48	80.3%	10.5%	9.2%

Fig. 1. The analysis times.

5 Experimental Evaluation

We show now the behaviour of our analyser on some benchmarks. We have used SWI-Prolog 3.3.2 over an AMD K5 100Mhz processor with 64Mbytes of memory, running Linux 2.2. The techniques of Subsection 4.2 have been applied.

In Figure 1, for every benchmark, we report its dimension in bytes, the time in seconds spent in the preprocessing phase (normalisation, abstraction and call graph construction), that spent for the fixpoint calculation, and the relative computational cost of the conjunction operation (Definition 3) w.r.t. the other operations of the domain and the shell (preprocessor) which normalises, abstracts and computes the fixpoint. As you can see, the preprocessing time is always small, while the fixpoint calculation is sometimes expensive and is much more related to the number of variables in the clauses of the program than to its dimension (compare the lucky case of `eliza.pl` with that of `heapify.pl`). Indeed, when that number becomes large, the time spent for the abstract conjunction explodes, as the fifth column shows. Thus a clever implementation of the conjunction is welcome.

We have compared our analyser with a goal-dependent analysis performed by using the *Sharing* $\times$ *Free* domain [12] inside the China analyser[4] [2]. The result is shown in Figure 2. The goal-dependent analysis is definitely more efficient, but must be re-executed for every query. W.r.t. precision, we have run some abstract queries with the goal-dependent analyser and we have compared the resulting abstract information with what we get by instantiating our goal-independent denotation on the queries.

In the third column, "`A`" means "`A` free", while "`(A,B,C)`" means that `A`, `B` and `C` are mutually independent, and is a compact notation for `(A,B)(A,C)(B,C)`. The last two columns show the results of the analysis with our analyser and with China, expressed in our domain. Our analyser is always as precise as China except for `app/6`, a version of `append/3` for incomplete lists.

[4] We thank Roberto Bagnara for his help with this experiment.

Benchmark	Predicate call	Input constraint	Our response	China's
ackermann.pl	`ackermann(A,B,C)`	`ABC(A,B,C)`	`(A,A)`	`(A,A)`
		`(B,B)`	`(A,A)(B,B) (C,C)`	`(A,A)(B,B) (C,C)`
append.pl	`append(A,B,C)`	`ABC(A,B,C)`	`B(A,B)`	`B(A,B)`
		`BC(A,B,C)`	`B(A,B)`	`B(A,B)`
		`C(A,B,C)`	`(A,B)`	`(A,B)`
		`(C,C)`	`(A,A)(B,B) (C,C)`	`(A,A)(B,B) (C,C)`
eliza.pl	`eliza(A)`	`A`	`true`	`true`
hanoi.pl	`hanoi(A,B,C,D,E)`	`ABCDE(A,B,C,D,E)`	`(A,A)`	`(A,A)`
		`(E,E)`	`(A,A)(B,B) (C,C)(E,E)`	`(A,A)(B,B) (C,C)(E,E)`
heapify.pl	`heapify(A,B)`	`AB(A,B)`	`true`	`true`
		`(A,A)`	`(A,A)(B,B)`	`(A,A)(B,B)`
map_coloring.pl	`color_map(A,B)`	`AB(A,B)`	`true`	`true`
queens.pl	`queens(A,B)`	`AB(A,B)`	`(A,A)(B,B)`	`(A,A)(B,B)`
		`(A,A)`	`(A,A)(B,B)`	`(A,A)(B,B)`
quicksort.pl	`quicksort(A,B)`	`AB(A,B)`	`(A,A)(B,B)`	`(A,A)(B,B)`
		`(A,A)`	`(A,A)(B,B)`	`(A,A)(B,B)`
openlist.pl	`nil(A,B)`	`AB(A,B)`	`B`	`B`
	`cons(A,B,C,D,E)`	`CDE(A,A)(B,D,E) (C,D,E)`	`CE(A,A)`	`CE(A,A)`
	`app(A,B,C,D,E,F)`	`BDEF(A,C,E,F) (A,D,E,F)(B,C,E,F) (B,D,E,F)`	`true`	`DF`
	`list2open(A,B,C)`	`BC(B,C)(A,A)`	`C(A,A)`	`C(A,A)`

Fig. 2. The comparison of our analysis with that done through China.

6 Conclusion

We have described the implementation of a static analyser based on the abstract domain for non pair-sharing and freeness of [13]. It shows that linear refinement can be used to devise practically useful domains. We do not know of any other implementation of a static analysis developed through linear refinement.

We have shown that reduction rules are necessary in order to obtain an efficient analysis. We have studied a sufficient condition which entails that a reduction rule does not introduce any loss of precision.

A promising widening operation would delete all the arrows whose dimension is too big. They are the cause of the computational cost of the analysis and are seldom useful in practice. Preliminary experiments have shown that a drastic performance improvement can be obtained.

Our analysis is almost always as precise as a traditional goal-dependent analysis. This justifies the research of more efficient implementations. At the same time, it challenges us to exploit the full power of the domain.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. Set-Sharing is Redundant for Pair-Sharing. In P. Van Hentenryck, editor, *Proc. of the 4th Int. Symp. on Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*, pages 53–67, Paris, France, 1997. Springer-Verlag, Berlin.
2. Roberto Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1997.
3. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19–20:149–197, 1994.
4. M. Bruynooghe, B. Demoen, D. Boulanger, M. Denecker, and A. Mulkers. A Freeness and Sharing Analysis of Logic Programs Based on a Pre-Interpretation. In R. Cousot and D. A. Schmidt, editors, *3rd Int. Symp. on Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 128–142, Aachen, Germany, 1996. Springer Verlag.
5. M. Codish and B. Demoen. Deriving Polymorphic Type Dependencies for Logic Programs Using Multiple Incarnations of Prop. In *Proc. of the first International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 281–296. Springer-Verlag, 1994.
6. M. Codish, V. Lagoon, and F. Bueno. An Algebraic Approach to Sharing Analysis of Logic Programs. *Journal of Logic Programming*, 42(2):110–149, February 2000.
7. M. Codish, H. Søndergaard, and P. J. Stuckey. Sharing and Groundness Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.
8. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.
9. P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
10. R. Giacobazzi and R. Ranzato. The Reduced Relative Power Operation on Abstract Domains. *Theoretical Comp. Science*, 216:159–211, 1999.
11. M. Hermenegildo, W. Warren, and S.K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programing*, 13(2 & 3):349–366, 1992.
12. D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, 1992.
13. Giorgio Levi and Fausto Spoto. Non Pair-Sharing and Freeness Analysis through Linear Refinement. In *Proc. of the Partial Evaluation and Program Manipulation Workshop*, pages 52–61, Boston, Mass., January 2000. ACM Press. Available at `http://strudel.di.unipi.it/papers`.
14. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
15. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables through Abstract Interpretation. In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 49–63, Paris, 1991. The MIT Press.
16. H. Søndergaard. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, 1986.

PROOFS

We recall from [13] the following definitions.

Definition 9. *Given $V \in \wp_f(\mathcal{V})$ and $\{v, v_1, v_2\} \subseteq V$, we define $(\mathbf{v_1}, \mathbf{v_2})_V = \{\exists_W c \in H_V \mid \mathsf{vars}(c(v_1)) \cap \mathsf{vars}(c(v_2)) = \emptyset\}$ and $\mathbf{v}_V = \{\exists_W c \in H_V \mid c(v) \in V \cup W\}$. When the set V is obvious from the context, we write $(\mathbf{v_1}, \mathbf{v_2})$ for $(\mathbf{v_1}, \mathbf{v_2})_V$ and $\mathbf{v}$ for $\mathbf{v}_V$. Given $\{(v_1^1, v_1^2), \ldots, (v_n^1, v_n^2), v_1, \ldots, v_m\}$, we write $(\mathbf{v_1^1}, \mathbf{v_1^2}) \ldots (\mathbf{v_n^1}, \mathbf{v_n^2})\mathbf{v_1} \ldots \mathbf{v_m}$ (the order is irrelevant) for $(\cap_{i=1,\ldots,n}(\mathbf{v_i^1}, \mathbf{v_i^2})) \cap (\cap_{i=1,\ldots,m}\mathbf{v_i})$.*

The linear arrow $\mathbf{l} \multimap \mathbf{r}$ is defined as

$$\mathbf{l} \multimap \mathbf{r} = \{h \in H_V \mid \textit{for every } h' \in \mathbf{l} \textit{ if } h \star^{H_V} h' \textit{ is defined then } h \star^{H_V} h' \in \mathbf{r}\}$$

for every $\mathbf{l}, \mathbf{r} \in \wp(H_V)$.

We define the concretisation map $\gamma_{Rep} : Abs_V \mapsto \wp(H_V)$ as

$$\gamma_{Rep}(A) = \bigcap_{l \Rightarrow r \in A} \mathbf{l} \multimap \mathbf{r}$$

for any $A \in Abs_V$.

The following definition formalises the intuitive concept of computation.

Definition 10. *A* computation *is whether an element of Abs_V, for any $V \in \wp_f(\mathcal{V})$, or a term of the form $\mathsf{op}(c_1, \ldots, c_n)$, where the c_i's are computations,* op *is the name of one of the abstract operators defined in Section 3 and its signature is respected. The evaluation of a computation is defined as $[\![A]\!] = A$ if $A \in Abs_V$ for some $V \in \wp_f(\mathcal{V})$ and $[\![\mathsf{op}(c_1, \ldots, c_n)]\!] = \mathsf{op}([\![c_1]\!], \ldots, [\![c_n]\!])$. Moreover, if ρ is a reduction rule, we define $[\![A]\!]^\rho = \rho_V(A)$ if $A \in Abs_V$ for some $V \in \wp_f(\mathcal{V})$ and $[\![\mathsf{op}(c_1, \ldots, c_n)]\!]^\rho = \rho_V(\mathsf{op}([\![c_1]\!]^\rho, \ldots, [\![c_n]\!]^\rho))$ if $\mathsf{op}([\![c_1]\!]^\rho, \ldots, [\![c_n]\!]^\rho) \in Abs_V$ for some $V \in \wp_f(\mathcal{V})$.*

*Proof (**Proof of Proposition 1**).* We have to prove that every computation c is such that $\mathsf{free}_V([\![c]\!]) \subseteq \mathsf{free}_V([\![c]\!]^\rho)$ and $\mathsf{nsh}_V([\![c]\!]) \subseteq \mathsf{nsh}_V([\![c]\!]^\rho)$. We prove that all the abstract operators of Section 3 are monotonic w.r.t. $\preceq$. This will entail the thesis by induction on c, since $\rho_V(A) \preceq A$ and it is easy to check that $A_1 \preceq A_2$ entails $\mathsf{free}_V(A_2) \subseteq \mathsf{free}_V(A_1)$ and $\mathsf{nsh}_V(A_2) \subseteq \mathsf{nsh}_V(A_1)$.

Assume $V \in \wp_f(\mathcal{V})$ and $A, A_1, A_2 \in Abs_V$ such that $A_1 \preceq A_2$.

We have trivially $\mathsf{rename}^{Abs_V}_{x \to n}(A_1) \preceq \mathsf{rename}^{Abs_V}_{x \to n}(A_2)$.

For $\mathsf{restrict}^{Abs_V}$, consider $l_2 \Rightarrow r \in \mathsf{restrict}^{Abs_V}_x(A_2)$. Then $l_2 = l_2' \setminus X$ with $l_2' \Rightarrow r \in A_2$, $(x, x) \notin l_2'$ and X as defined in Definition 4. Then there exists $l_1' \Rightarrow r \in A_1$ such that $l_1' \subseteq l_2' \cup \{(v, v') \mid v, v' \in V,\ v \neq v',\ (v, v) \in l_2'\}$. Since $(x, x) \notin l_2'$, we have $(x, x) \notin l_1'$. Therefore, $l_1' \setminus X \Rightarrow r \in \mathsf{restrict}^{Abs_V}_x(A_1)$, and $l_1' \setminus X \subseteq l_2' \cup \{(v, v') \mid v, v' \in V,\ v \neq v',\ (v, v) \in l_2'\} \setminus X = (l_2' \setminus X) \cup \{(v, v') \mid v, v' \in V \setminus x,\ v \neq v',\ (v, v) \in l_2' \setminus X\}$, since $(x, x) \notin l_2'$.

For expand^{Abs_V}, let $l_2 \Rightarrow r \in \mathsf{expand}^{Abs_V}_n(A_2)$ with $l_2 \Rightarrow r \in A_2$. Then there exists $l_1 \Rightarrow r \in A_1$ with $l_1 \subseteq l_2 \cup \{(v, v') \mid v, v' \in V,\ v \neq v',\ (v, v) \in l_2\}$ and $l_1 \Rightarrow r \in \mathsf{expand}^{Abs_V}_n(A_1)$. If $l_2[n/?_1][?1/?2] \Rightarrow r[n/?_1][?1/?2] \in \mathsf{expand}^{Abs_V}_n(A_2)$

with $l_2 \Rightarrow r \in A_2$, then there exists $l_1 \Rightarrow r \in A_1$ with $l_1 \subseteq l_2 \cup \{(v,v') \mid v,v' \in V,\ v \neq v',\ (v,v) \in l_2\}$. Therefore, $l_1[n/?_1][?1/?2] \subseteq l_2[n/?_1][?1/?2] \cup \{(v,v') \mid v,v' \in V,\ v \neq v',\ (v,v) \in l_2\}[n/?_1][?1/?2] \subseteq l_2[n/?_1][?1/?2] \cup \{(v,v') \mid v,v' \in V \cup n,\ v \neq v',\ (v,v) \in l_2[n/?_1][?1/?2]\}$. Since $l_1[n/?_1][?1/?2] \Rightarrow r[n/?_1][?1/?2] \in \mathsf{expand}_n^{Abs_V}(A_1)$, we have the thesis.

For $\cup^{Abs_V}$, consider $l_2 l \Rightarrow r \in \cup^{Abs_V}(A_2, A)$, with $l_2 \Rightarrow r \in A_2$ and $l \Rightarrow r \in A$. There exists $l_1 \Rightarrow r \in A_1$ with $l_1 \subseteq l_2 \cup \{(v,v') \mid v,v' \in V,\ v \neq v',\ (v,v) \in l_2\}$. Therefore, $l_1 l \subseteq l_2 l \cup \{(v,v') \mid v,v' \in V,\ v \neq v',\ (v,v) \in l_2\} \subseteq l_2 l \cup \{(v,v') \mid v,v' \in V,\ v \neq v',\ (v,v) \in l_2 l\}$. Since $l_1 l \Rightarrow r \in \cup^{Abs_V}(A_1, A)$ and the same argument can be used for the symmetrical case of the definition of $\cup^{Abs_V}$, we have the thesis.

For $\star^{Abs_V}$, consider $l_1 \cdots l_n \Rightarrow r \in A_2 \star^{Abs_V} A$, with $r_1 \cdots r_n \Rightarrow r \in A \cup T$ (T is the set of Definition 3), $l_i \Rightarrow r'_i \in A_2 \cup T$ and $\mathbf{r'_i} \subseteq \mathbf{r_i}$ for $i = 1, \ldots, n$. Then $l'_i \Rightarrow r'_i \in A_1 \cup T$ with $l'_i \subseteq l_i \cup \{(v,v') \mid v,v' \in V,\ v \neq v',\ (v,v) \in l_i\}$ and $l'_1 \cdots l'_n \Rightarrow r \in A_1 \star^{Abs_V} A$ with $l'_1 \cdots l'_n \subseteq l_1 \cdots l_n \cup \{(v,v') \mid v,v' \in V,\ v \neq v',\ (v,v) \in l_1 \cdots l_n\}$. Consider now $l_1 \cdots l_n \Rightarrow r \in A_2 \star^{Abs_V} A$ with $r_1 \cdots r_n \Rightarrow r \in A_2 \cup T$, $l_i \Rightarrow r'_i \in A \cup T$ and $\mathbf{r'_i} \subseteq \mathbf{r_i}$. There exists $l' \Rightarrow r \in A_1 \cup T$ such that $l' \subseteq r_1 \cdots r_n \cup \{(v,v') \mid v,v' \in V,\ v \neq v', (v,v) \in r_1 \cdots r_n\}$. Given $k \in l'$, whether $k \in r_1 \cdots r_n$ or $k = (v,v')$ with $(v,v) \in r_1 \cdots r_n$. In both cases there exists i, $1 \leq i \leq n$, such that $\mathbf{r'_i} \subseteq \mathbf{r_i} \subseteq \mathbf{k}$, and we can select a set S of natural numbers between 1 and n such that $\cup_{i \in S} l_i \Rightarrow r \in A_1 \star^{Abs_V} A$ and $\cup_{i \in S} l_i \subseteq l_1 \cdots l_n \subseteq l_1 \cdots l_n \cup \{(v,v') \mid v,v' \in V,\ v \neq v',\ (v,v) \in l_1 \cdots l_n\}$. The other case of $\star^{Abs_V}$ is symmetrical.

Given $V \in \wp_f(\mathcal{V})$ and two arrows $l \Rightarrow r$ and $l' \Rightarrow r'$, we write $l \Rightarrow r \preceq l' \Rightarrow r'$ if and only if $r = r'$ and $l \subseteq l' \cup \{(v,v') \mid v,v' \in V, v \neq v', (v,v) \in l'\}$. The relation $\preceq$ turns out to be a well founded partial order. It is obvious that $A \preceq A'$ if and only if for each arrow $a' \in A'$ there exists an arrow $a \in A$ such that $a \preceq a'$.

*Proof (**Proof of Proposition 2**).* Given an arrow $a \in A$, let us consider the set of all the arrows $b \in A$ with $b \preceq a$, which we denote by $\downarrow a$. If $A' \preceq A$, the least element of $\downarrow a$, namely $\bigcap \downarrow a$ has to be in A'. It turns out that $\bar{A} = \{\bigcap(\downarrow a) \mid a \in A\}$ is the least subset of A according to the $\preceq$ ordering. It is easy to check that $\bar{A} = \rho^1(A)$.

Now, we want to prove that $\gamma_{Rep}(\rho^1(A)) = \gamma_{Rep}(A)$. It is enough to prove that, given two arrows a and a', if $a \preceq a'$ then $\gamma_{Rep}(a) \subseteq \gamma_{Rep}(a')$. If $a \preceq a'$, then $a = l \Rightarrow r$, $a' = l' \Rightarrow r$ and $\gamma_{Rep}(l) \supseteq \gamma_{Rep}(l')$. By properties of the linear refinement, $\gamma_{Rep}(a) \subseteq \gamma_{Rep}(a')$ follows.

*Proof (**Proof of Proposition 3**).* Given $V \in \wp_f(\mathcal{V})$, the map ρ^2_V, applied to $A \in Abs_V$, removes (v,v') from the left hand side of $l \Rightarrow r$ if and only if $(v,v) \in l$ and $v \neq v'$. Therefore it is reductive w.r.t. $\preceq$. Moreover, $\mathsf{dim}(\rho^2_V(A)) \leq \mathsf{dim}(A)$ and $\gamma_{Rep}(A) = \gamma_{Rep}(\rho^2_V(A))$. Indeed, given $h \in H_V$, if $h \in (\mathbf{v}, \mathbf{v})$ then h is ground and $h \in (\mathbf{v}, \mathbf{v'})$ for any $v' \in V$.