Freeness Analysis Through Linear Refinement *

Patricia Hill[1] and Fausto Spoto[2]**

[1] School of Computer Studies, University of Leeds, Leeds, LS2 9JT, UK
hill@scs.leeds.ac.uk
[2] Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56100 Pisa, Italy
spoto@di.unipi.it
Ph.: +39-050-887248 Fax: +39-050-887226

Abstract Linear refinement is a technique for systematically constructing abstract domains for program analysis directly from a basic domain representing just the property of interest. This paper for the first time uses linear refinement to construct a new domain instead of just reconstructing an existing one. This domain is designed for definite freeness analysis and consists of sets of dependencies between sets of variables. We provide explicit definitions of the operations for this domain which we show to be safe with respect to the concrete operations. We illustrate how the domain may be used in a practical analysis by means of a small example.

Keywords: *Abstract interpretation, abstract domain, linear refinement, static analysis, freeness analysis, logic programming.*

1 Introduction

Linear refinement [13] is a technique for systematically constructing abstract domains for program analysis. Given a basic abstract domain representing just the property of interest together with an appropriate concrete operation (which, as we are analysing logic programs, is usually unification) the new more refined domain is constructed.

Freeness analysis is concerned with the computation of a set of variables (called *free* variables) which, at a given program point, are guaranteed to be only bound to variables. As pointed out in [20], the information collected by a freeness analysis is useful for increasing the power of non strict independent AND-parallelism [14], for optimising unification, for goal ordering and for the avoidance of type checking. Moreover, for sharing analysis (which deals with the possible sharing of terms among variables), freeness is an important component that can improve both the efficiency and the precision of the analyser.

* Part of this work was supported by EPSRC grant GR/M05645.
** Part of this work was done while Fausto Spoto was visiting the School of Computer Studies of the University of Leeds, UK.

Freeness analysis has received relatively little attention from the research community, compared for instance with groundness analysis [1, 6]. If we consider the freeness domains proposed in the literature, we find that two different approaches are taken. The first considers freeness as a *mode* [9, 20]. In this case, a substitution is simply abstracted as the set of free variables (that is, those which are mapped by the substitution to variables). This basic approach is extremely imprecise. To improve precision, [9] combines modes with the information about the set of variables present in the term bound to each variable, whereas [20] combines freeness with sharing analysis, to improve both the sharing and the freeness components. Both [9] and [20] show that the basic freeness abstraction is not acceptable at all. This is because even if it is known that a variable is free in the substitutions θ_1 and θ_2, there is no guarantee that it will be free in their most general unifier. Then the most precise abstraction of unification is so imprecise to be useless. Compare this with the case of groundness, where the simple abstraction of a substitution into the set of variables that are ground allows the definition of an abstract unification operation which is imprecise but still useful. The second approach abstracts a substitution into an abstract set of equations, which can be seen as an approximate description of the concrete set of equations represented by the substitution itself [3, 4, 5, 19, 23]. This means that part of the functor structure of the substitution is maintained in the abstract sets of equations, resulting in an extremely precise analysis but with rather complex abstract operations. In [15] an intermediate approach is proposed. Here the terms are fully abstracted apart from the positions of their variables. These are maintained by means of *paths* that define their positions in the term.

The contribution of this paper is in the definition of a new domain for definite freeness analysis which is able to express freeness dependencies without the help of any auxiliary domain (like sharing, for instance). This domain is constructed by means of a linear refinement operation, as done in the case of groundness [21] and types [17]. We provide a representation for this domain and safe operations on this representation. It can be shown that our domain is contained in the $\mathit{Sharing} \times \mathit{Free}$ domain of [16]. This means that its precision is no more than that of [16]. The advantage of our domain is its natural definition, together with the simplicity of its representation and its abstract operators. Note that, previously, only a restricted form of linear refinement (called the Heyting refinement) has been used. It was applied to the reconstruction of the domain *Pos* for groundness analysis [21] and the construction of type domains [17].

The paper is organised as follows. Section 2 provides some preliminary definitions. Section 3 defines the concrete domain and its abstraction into a basic domain for freeness analysis. Section 4 refines this basic domain through the standard approach for linear refinement and shows that this does not lead to interesting domains. Section 5 shows how to overcome this problem through the use of *internal dependencies*. Section 6 defines a representation of our domain of internal dependencies based on sets of dependencies between variables. Sections 7 and 8 define the algorithms for the computation of the abstraction function and of the abstract operators on the representation, respectively. Section 9 provides an example of freeness analysis of logic programs based on our domain. Section 10 summarises the contributions of this paper, compares our work with the standard framework for freeness analysis, which is based on the *Sharing* $\times$ *Free* domain, and consider the problem of condensing for our domain.

2 Preliminaries

2.1 Terms and substitutions

Given a set of variables V and a set of function symbols Σ with associated arity, we define $\mathrm{terms}(\Sigma, V)$ as the minimal set of terms built from V and Σ as: $v \in \mathrm{terms}(\Sigma, V)$ for every $v \in V$ and if $t_1, \ldots, t_n \in \mathrm{terms}(\Sigma, V)$ and $\mathtt{f} \in \Sigma$ is a function symbol of arity $n \geq 0$, then $\mathtt{f}(t_1, \ldots, t_n) \in \mathrm{terms}(\Sigma, V)$. $\mathrm{vars}(t)$ is the set of variables occurring in t. A term t is ground if $\mathrm{vars}(t) = \emptyset$. Given a set of variables V and a variable x, $V \cup x$ means $V \cup \{x\}$ and $V - x$ means $V - \{x\}$.

A substitution θ is a map from variables into terms. We define the sets $\mathrm{dom}(\theta) = \{x | \theta(x) \neq x\}$ and $\mathrm{rng}(\theta) = \cup_{x \in \mathrm{dom}(\theta)} \mathrm{vars}(\theta(x))$. We require $\mathrm{dom}(\theta)$ to be finite. We define Θ_V as the set of idempotent substitutions θ such that $\mathrm{dom}(\theta) \cup \mathrm{rng}(\theta) \subseteq V$ and $\mathrm{dom}(\theta) \cap \mathrm{rng}(\theta) = \emptyset$. Given θ and a set of variables R, we define $\theta|_R$ as $\theta|_R(x) = \theta(x)$ if $x \in R$ and $\theta|_R(x) = x$ otherwise. Given a term $t \in \mathrm{terms}(\Sigma, V)$ and $\theta \in \Theta_V$, $t\theta \in \mathrm{terms}(\Sigma, V)$ is the term obtained with parallel substitution of every variable x in t with $\theta(x)$.

$\mathcal{P}(S)$ is the powerset of a set S. $\mathcal{P}_f(S)$ is the set of all *finite* subsets of S.

2.2 The domain of existential Herbrand constraints

Let $\mathcal{V}$ be an infinite set of variables. For each $V \in \mathcal{P}_f(\mathcal{V})$, we have the set of Herbrand equations

$$C_V = \{t^1 = t^2 | t^1, t^2 \in \mathrm{terms}(\Sigma, V)\} .$$

Suppose $c \in C_V$. Then we say *$c\theta$ is true* if $t^1\theta = t^2\theta$ *for every* $(t^1 = t^2) \in c$. We know [18] that if there exists $\theta \in \Theta_V$ such that $c\theta$ is true, then c can be put in *normal form* $\mathrm{mgu}(c) = \{{t'}^1_j = {t'}^2_j | j \in J\}$, where ${t'}^1_j \in V$ are distinct variables which do not occur in ${t'}^2_k$ for every $j, k \in J$ and the variables in $mgu(c)$ are all contained in the variables of c. Moreover, we have $c\theta$ is true if and only if $\mathrm{mgu}(c)\theta$ is true. If no θ exists such that $c\theta$ is true, then $\mathrm{mgu}(c)$ and hence the normal form of c are undefined. If c is in normal form, c can be seen as a substitution, and we use the notation $c(v)$ meaning the term t on the right of an equality $v = t \in c$ if such an equality exists, v itself otherwise.

Let $\mathcal{V}$ and $\mathcal{W}$ be disjoint infinite sets of variables. For each $V \in \mathcal{P}_f(\mathcal{V})$, we have a set of constraints, called existential Herbrand constraints:

$$H_V = \{\exists_W c \,|W \in \mathcal{P}_f(\mathcal{W}), c \in \mathcal{P}_f(C_{V \cup W})\} .$$

Here, $\mathcal{V}$ are called the program variables and $\mathcal{W}$, the existential variables. Given a Herbrand constraint $h = \exists_W c \in H_V$, the set of its solutions is defined as

$$\mathrm{sol}_V(h) = \{\theta|_V \,|\theta \in \Theta_{V\cup W},\ \mathrm{rng}(\theta) \subseteq V \text{ and } c\theta \text{ is true}\} \ .$$

Hence $\mathrm{sol}_V(\exists_W c) = \mathrm{sol}_V(\exists_W \mathrm{mgu}(c))$. $\exists_W c$ is said to be in *normal form* if c is in normal form.

Two constraints $h_1, h_2 \in H_V$ are called *equivalent* if and only if $\mathrm{sol}_V(h_1) = \mathrm{sol}_V(h_2)$. In the following, a constraint will stand for its equivalence class. Since, as shown above, every consistent constraint can be put in an equivalent normal form, in the following we will consider only normal constraints.

Two operations and diagonal elements are defined on (normal) Herbrand constraints:

Definition 1 (Conjunction on H_V). *Assuming $W_1 \cap W_2 = \emptyset$* [1]*, we have:*

$$(\exists_{W_1} c_1) \star^{H_V} (\exists_{W_2} c_2) = \begin{cases} \exists_{W_1\cup W_2} \mathrm{mgu}(c_1 \cup c_2) & \textit{if } \mathrm{mgu}(c_1 \cup c_2) \textit{ exists} \\ \textit{undefined} & \textit{otherwise.} \end{cases}$$

Definition 2 (Restriction on H_V). *We define the restriction of a Herbrand constraint $\exists_W c$ with respect to a program variable x as*

$$\exists_x^{H_V}(\exists_W c) = \exists_{W\cup N} c[N/x] \qquad \textit{where } N \notin (V \cup W).$$

That is, we consider the program variable x as a new existential variable N.

Definition 3 (Diagonal elements on H_V). *Given two sequences of distinct variables $\langle x_1, \dots, x_n\rangle$ and $\langle y_1, \dots, y_n\rangle$ in V whose intersection is empty, we define*

$$\delta^{H_V}_{\langle x_1,\dots,x_n\rangle,\langle y_1,\dots,y_n\rangle} = \{x_i = y_i | i = 1, \dots, n\} \ .$$

2.3 Abstract interpretation

We briefly recall the main result about abstract interpretation [8]. Let $(\mathcal{C}, \preceq)$ (concrete domain) and $(\mathcal{A}, \leq)$ (abstract domain) be two posets. A Galois connection between them is a pair of functions α (*abstraction*) and γ (*concretisation*) such that

1. α and γ are monotonic,
2. for all $x \in \mathcal{C}$, we have $x \preceq (\gamma \circ \alpha)(x)$ and
3. for all $y \in \mathcal{A}$, we have $(\alpha \circ \gamma)(y) \leq y$.

[1] Note that this is not restrictive because the names of existential variables are irrelevant: given a Herbrand constraint $\exists_W c$, the constraint $\exists_{W'} c[W'/W]$ is equivalent to it. Hence we can always assume Herbrand constraints to be renamed apart with respect to existential variables.

A Galois connection is called a Galois insertion if $(\alpha \circ \gamma)(y) = y$. In a Galois insertion the abstract domain does not contain useless elements, i.e., elements which are not the image through α of some concrete element.

Let $f : \mathcal{C}^n \to \mathcal{C}$ be a concrete operator and assume that $\tilde{f} : \mathcal{A}^n \to \mathcal{A}$. Then $\tilde{f}$ is correct with respect to f if and only if for all $x_1, \ldots, x_n \in \mathcal{A}$ we have $\alpha(f(\gamma(x_1), \ldots, \gamma(x_n))) \preceq \tilde{f}(x_1, \ldots, x_n)$. For each operator f, there exists an optimal (most precise) correct abstract operator $\tilde{f}$ defined as $\tilde{f}(y_1, \ldots, y_n) = \alpha(f(\gamma(y_1), \ldots, \gamma(y_n)))$, where α is extended to sets $S \in \mathcal{C}$ by defining $\alpha(S) = \wedge_{s \in S} \alpha(s)$. Note that this requires $\mathcal{A}$ to be completely $\wedge$–closed and to contain a top element, i.e., to be a Moore family. In general, if X is a subset of a lattice L then $\curlywedge(X)$ denotes the Moore–closure of X, i.e., the least subset of L containing X which is a Moore family of L.

2.4 Domain refinement

A systematic approach to the development of abstract domains is based on the use of domain refinement operators. Given an abstract domain $\mathcal{A}$, a domain refinement operator R yields an abstract domain $R(\mathcal{A})$ which is more precise than $\mathcal{A}$. Classical domain refinement operators are reduced product and disjunctive completion [11]. The reduced product of two domains $\mathcal{A}$ and $\mathcal{B}$ is isomorphic to the Cartesian product of $\mathcal{A}$ and $\mathcal{B}$, modulo the equivalence relation $\langle a_1, b_1 \rangle \equiv \langle a_2, b_2 \rangle$ if and only if $a_1 \wedge b_1 = a_2 \wedge b_2$. This means that pairs having the same *meaning* are identified.

Linear completion was proposed in [12] as a powerful domain refinement operator. It allows to include in a domain the information relative to propagation of the abstract property of interest before and after the application of a concrete operator $\boxtimes$.

Let L be a complete lattice and $\boxtimes$ be a partial map $\boxtimes : L \times L \to L$. The functions $\multimap^{\boxtimes}$ and $\multimapinv^{\boxtimes}$ are defined as

$$\begin{aligned} a \multimap^{\boxtimes} b &= \bigvee_L \{l \in L | \text{if } a \boxtimes l \text{ is defined then } a \boxtimes l \leq_L b\} \ , \\ a \multimapinv^{\boxtimes} b &= \bigvee_L \{l \in L | \text{if } l \boxtimes a \text{ is defined then } l \boxtimes a \leq_L b\} \ . \end{aligned} \tag{1}$$

Note that if $\boxtimes$ is commutative then $a \multimapinv^{\boxtimes} b = a \multimap^{\boxtimes} b$ for every $a, b \in L$.

We define the linear refinement domain $L \multimap^{\boxtimes} L$ as

$$L \multimap^{\boxtimes} L = L \sqcap \curlywedge \{a \multimap^{\boxtimes} b, a \multimapinv^{\boxtimes} b | a, b \in L\} \ ,$$

which can be simplified into

$$L \multimap^{\boxtimes} L = \curlywedge \{a \multimap^{\boxtimes} b, a \multimapinv^{\boxtimes} b | a, b \in L\}$$

if the elements of L can be obtained as (greatest lower bounds of) arrows. This case is important since it allows a simpler representation of the domain and simpler operations on this representation.

The right linear refinement of L is defined as

$$L \multimap_r^{\boxtimes} L = L \sqcap \curlywedge \{a \multimap^{\boxtimes} b | a, b \in L\} \ . \tag{2}$$

Again, if the elements of L can be obtained as (greatest lower bounds of) arrows, Equation (2) can be simplified into

$$L \multimap_r^{\boxtimes} L = \curlywedge \{a \multimap^{\boxtimes} b | a, b \in L\} \ . \tag{3}$$

When not explicitly stated, $L \multimap^{\boxtimes} L$ will stand for $L \multimap_r^{\boxtimes} L$.

3 Freeness analysis

The concrete domain we abstract is the collecting version [8] of the domain of Herbrand constraints introduced in Subsection 2.2, i.e., the lattice $\langle \mathcal{P}(H_V), \cap, \cup, H_V, \emptyset \rangle$. The operations and the diagonal elements on H_V are point–wise extended to $\mathcal{P}(H_V)$ as follows:

$$\begin{aligned} S_1 \star^{\mathcal{P}(H_V)} S_2 &= \left\{ h_1 \star^{H_V} h_2 \,\middle|\, h_1 \in S_1,\ h_2 \in S_2 \text{ and } h_1 \star^{H_V} h_2 \text{ is defined} \right\} \\ \exists_x^{\mathcal{P}(H_V)} S &= \left\{ \exists_x^{H_V} h \,\middle|\, h \in S \right\} \qquad \delta_{\tilde{x},\tilde{y}}^{\mathcal{P}(H_V)} = \left\{ \delta_{\tilde{x},\tilde{y}}^{H_V} \right\} \ . \end{aligned}$$

The first abstraction of this domain is a domain able to model basic freeness and nothing else.

Given a finite set of variables V, we define the set of variables that are free in a Herbrand constraint as

$$\text{free}\,(\exists_W c) = \{v \in V \,| c(v) \in (\mathcal{V} \cup \mathcal{W})\} \ .$$

The set of Herbrand constraints such that the variables U are free is defined as

$$\text{free}(U) = \{\, h \in H_V \,|\, U \subseteq \text{free}(h)\} \ .$$

The basic abstract domain for freeness analysis is

$$\mathit{Free}_V^0 = \{\text{free}(U) | U \subseteq V\} \ ,$$

ordered with respect to set inclusion. We have that $\text{free}(\emptyset)$ is the whole set of Herbrand constraints, and $\text{free}(V_1) \cap \text{free}(V_2) = \text{free}(V_1 \cup V_2)$ for every $V_1, V_2 \subseteq V$. Then Free_V^0 is a Moore family of $\mathcal{P}(H_V)$.

From now on, capital letters will stand for sets of variables, small letters for single variables and bold face letters for elements of the domain, originated by the corresponding variables. For instance, we will write $\mathbf{U}$ for $\text{free}(U)$ and $\mathbf{xyz}$ for $\text{free}(\{x, y, z\})$. $\emptyset$ will stand for $\text{free}(\emptyset)$, but we will write explicitly $\text{free}(\emptyset)$ when this notation can give rise to ambiguity. We write $\mathbf{U} \setminus U'$ for $\text{free}(U \setminus U')$ for every sets of variables U and U'.

Consider $c = \{x = \mathtt{f}(y), z = w\}$ and $V = \{k, x, y, w, z\}$. We have $c \in \mathbf{kywz}$, but $c \notin \mathbf{kxywz}$.

Elements of $\mathcal{P}(H_V)$ can be mapped into Free_V^0 by the abstraction map α_V^0 which is an upper closure operator:

$$\alpha_V^0(S) = \bigcap\{f \in \mathit{Free}_V^0 | S \subseteq f\} .$$

The concretisation map γ_V^0 is the identity map. $\langle \alpha_V^0, \gamma_V^0 \rangle$ is a Galois insertion from $\mathcal{P}(H_V)$ into Free_V^0.

4 A naive approach to linear refinement

Consider the two Herbrand constraints $c_1 = \{x = \mathtt{a}\}$ and $c_2 = \{y = \mathtt{a}\}$ over $V = \{x, y, z\}$. Their unification leaves z free. A useful domain for freeness analysis must capture this behaviour. In Free_V^0, c_1 is abstracted into $\mathbf{yz}$ and c_2 into $\mathbf{xz}$. Let $\star^{\mathit{Free}_V^0}$ be the best correct approximation of $\star^{\mathcal{P}(H_V)}$. We have $\mathbf{yz} \star^{\mathit{Free}_V^0} \mathbf{xz} = \mathrm{free}(\emptyset)$ which is not contained in $\mathbf{z}$ (since $\{x = \mathtt{a}, y = z\}$ is abstracted into $\mathbf{yz}$ as c_1, and its unification with c_2 leaves z non free). This result means that Free_V^0 is useless for freeness analysis.

It seems reasonable to use $\boxtimes = \star^{\mathcal{P}(H_V)}$ in the linear refinement definition of Subsection 2.4, in order to improve the Free_V^0 domain with freeness dependencies information. The operation $\vee$ is set union and the ordering $\leq$ is set inclusion. We obtain the domains

$$\mathit{Fr}_V^0 = \mathit{Free}_V^0 \qquad \mathit{Fr}_V^{i+1} = \mathit{Fr}_V^i \multimap^{\star^{\mathcal{P}(H_V)}} \mathit{Fr}_V^i \quad \text{for } i \geq 1.$$

Consider Fr_V^1. The constraint c_1 is contained in $\mathbf{yz}$. We want to show that $\mathbf{yz}$ is the best approximation of c_1 in Fr_V^1. Indeed, if $c_1 \in \mathbf{L} \multimap \mathbf{R}$ and $\mathbf{L} \multimap \mathbf{R}$ is not top, then $R \neq \emptyset$. Since $c_3 = \{x = y, z = y\}$ belongs to every element of Free_V^0 and hence to L, we would have $c_3 \star^{\mathcal{P}(H_V)} c_1 \in R$, which is a contradiction since $R \neq \emptyset$ and all three variables are made non free by the unification. Then c_1 is abstracted into $\mathbf{yz}$ in Fr_V^1. By symmetry, c_2 is abstracted into $\mathbf{xz}$ in Fr_V^1. The same argument used above in the case of Free_V^0 shows that the best correct approximation of the $\star^{\mathcal{P}(H_V)}$ operation is not contained in $\mathbf{z}$ when applied to c_1 and c_2. Again, Fr_V^1 is useless in practice for freeness analysis. Moreover, we must use Equation (2) rather than Equation (3), since the elements of Free_V^0 cannot be obtained as arrows. This means that Fr_V^1 is a complex domain, formed by elements of Free_V^0 and new elements obtained as arrows. Devising a representation for Fr_V^1 and correct (and possibly optimal) operators on this representations would not be an easy task.

Consider Fr_V^2. This domain *might* be able to conclude that the unification of c_1 and c_2 leaves z free. However, it is extremely complex. Devising a representation with correct (and possibly optimal) operators becomes almost infeasible. The resulting operators would be computationally expensive.

These arguments show that this is not the refinement we were looking for. In the following, we will define a different freeness domain as linear refinement of the domain Free_V^0 where, however, the operation $\star^{\mathcal{P}(H_V)}$ is replaced by a new operation that protects local variables.

5 Internal freeness dependencies

Consider the constraint $c = \{y = \mathtt{f}(\mathtt{a})\}$. We cannot be sure that x is still free after conjunction with a constraint c' where even both x and y are free. However, if x is free in c', instantiation of x in $\mathrm{mgu}(c, c')$ can only be a consequence of the non-freeness of y in c which pushes c' to instantiate x. Then, *from the point of view of* c, it is an *external* mechanism. Namely, c instantiate y and c' *replies* by instantiating x in turn.

Assume we want to model the effects *internal* to c of the conjunction with c'. For instance, in the conjunction of the constraint $c = \{y = \mathtt{f}(\mathtt{a}), x = v, h = w\}$ with $c' = \{y = \mathtt{f}(x), w = \mathtt{a}\}$, we want to model the instantiation of w and h, but not the instantiation of x and v, which are an external instantiation and a consequence of an external instantiation from the point of view of c. This can be achieved by refining the Free^0_V domain with respect to the operation

$$\mathrm{internal}^{\mathcal{P}(H_V)}(p_1, p_2) = \left\{ \mathrm{internal}^{H_V}(h_1, h_2) \;\middle|\; \begin{array}{l} h_1 \in p_1,\ h_2 \in p_2 \text{ and} \\ \mathrm{internal}^{H_V}(h_1, h_2) \text{ is defined} \end{array} \right\},$$

for every $p_1, p_2 \in \mathcal{P}(H_V)$, where $\mathrm{internal}^{H_V} = \lambda h, h'.h \star^{H_V} \mathrm{over}(h')$ and $\mathrm{over}(h')$ is h' where all free variables have been renamed into new, overlined, variables. Formally, we define $\mathrm{over}(\exists_W c) = \exists_{\mathrm{vars}(\mathrm{over}(c)) \setminus V} \mathrm{over}(c)$, where $\mathrm{over}(\{x_1 = t_1, \ldots, x_n = t_n\}) = \{\mathrm{over}(x_1 = t_1), \ldots, \mathrm{over}(x_n = t_n)\}$ and

$$\mathrm{over}(x_1 = t_1) = \begin{cases} \mathrm{over}(x_1) = \mathrm{over}(t_1) & \text{if } t_1 \in \mathcal{V} \cup \mathcal{W} \\ x_1 = \mathrm{over}(t_1) & \text{otherwise} \end{cases}$$

$$\mathrm{over}(\mathtt{f}(t_1, \ldots, t_m)) = \mathtt{f}(\mathrm{over}(t_1), \ldots, \mathrm{over}(t_m)) \qquad \mathrm{over}(v) = \begin{cases} \overline{v} & \text{if } v \in \mathcal{V} \\ v & \text{if } v \in \mathcal{W}. \end{cases}$$

For instance, $\mathrm{over}(\{y = \mathtt{f}(\mathtt{a}), x = v, h = w\}) = \exists_{\overline{x}, \overline{v}, \overline{h}, \overline{w}}\{y = \mathtt{f}(\mathtt{a}), \overline{x} = \overline{v}, \overline{h} = \overline{w}\}$ and $\mathrm{over}(\exists_{\{w\}}\{x = \mathtt{f}(w), y = w, h = \mathtt{g}(z)\}) = \exists_{\{w, \overline{y}, \overline{z}\}}\{x = \mathtt{f}(w), \overline{y} = w, h = \mathtt{g}(\overline{z})\}$.

By computing $\mathrm{mgu}(c, \mathrm{over}(c'))$, we can observe in isolation the internal and immediate effects of c' on c, without bothering about the fact that these effects can reach c', inducing some new instantiations that can reach c in turn, and so on.

The expansion of the right arrow $\twoheadrightarrow$ of Equation (1) yields in our case (note that $\mathrm{internal}^{\mathcal{P}(H_V)}$ is a total map, while $\star^{H_V}$ is not):

$$\begin{aligned} \mathbf{L} \twoheadrightarrow^{\mathrm{internal}^{\mathcal{P}(H_V)}} \mathbf{R} &= \bigvee_{\mathcal{P}(H_V)} \left\{ p \in \mathcal{P}(H_V) \;\middle|\; \mathrm{internal}^{\mathcal{P}(H_V)}(p, \mathbf{L}) \leq_{\mathcal{P}(H_V)} \mathbf{R} \right\} \\ &= \bigcup \left\{ p \in \mathcal{P}(H_V) \;\middle|\; \mathrm{internal}^{\mathcal{P}(H_V)}(p, \mathbf{L}) \subseteq \mathbf{R} \right\} \\ &= \left\{ h \in H_V \;\middle|\; \begin{array}{l} \text{for all } h' \in H_V, \\ \text{if } h' \in \mathbf{L} \text{ and } h \star^{H_V} \mathrm{over}(h') \text{ is defined} \\ \text{then } h \star^{H_V} \mathrm{over}(h') \in \mathbf{R} \end{array} \right\}. \end{aligned}$$

These kinds of dependencies will be called internal dependencies and can be seen as the building blocks of the unsatisfactory, global dependencies of the previous section.

We define the refinement:

$$Free^1_V = Free^0_V \rightarrowtriangle^{\text{internal}^{\mathcal{P}(H_V)}} Free^0_V\ .$$

Note that every point $\mathbf{U}$ of $Free^0_V$ is such that $\mathbf{U} = \mathbf{V} \rightarrowtriangle^{\text{internal}^{\mathcal{P}(H_V)}} \mathbf{U}$. Then we can use Equation (3) instead of Equation (2).

From now on, $\rightarrowtriangle$ will stand for $\rightarrowtriangle^{\text{internal}^{\mathcal{P}(H_V)}}$.

$Free^1_{\{x,y\}}$ is depicted in Figure 1. $\emptyset \rightarrowtriangle \mathbf{x}$ contains no constraint. $\mathbf{xy} \rightarrowtriangle \mathbf{xy} \cap \mathbf{x} \rightarrowtriangle \mathbf{x} = \mathbf{xy} \rightarrowtriangle \mathbf{xy} \cap \mathbf{y} \rightarrowtriangle \mathbf{y} = \mathbf{x} \rightarrowtriangle \mathbf{x} \cap \mathbf{y} \rightarrowtriangle \mathbf{y} = \{\varepsilon\}$. $\mathbf{xy} \rightarrowtriangle \mathbf{xy} = \{\varepsilon, \{x = y\}\}$. $\mathbf{x} \rightarrowtriangle \mathbf{x}$ contains ε and constraints like $\{y = \mathtt{f}(\mathtt{a})\}$ or $\exists_{\{w\}}\{y = \mathtt{f}(w)\}$. $\mathbf{xy} \rightarrowtriangle \mathbf{x}$ contains the constraints above, $\{x = y\}$ and constraints like $\{y = \mathtt{f}(x)\}$. Symmetrically for $\mathbf{y} \rightarrowtriangle \mathbf{y}$ and $\mathbf{xy} \rightarrowtriangle \mathbf{y}$. The top element contains the whole set of constraints. Namely, it is the only point containing constraints like $\{x = \mathtt{f}(\mathtt{a}), y = \mathtt{f}(\mathtt{f}(\mathtt{a}))\}$. This means that the abstraction of $\{x = \mathtt{f}(\mathtt{a})\}$ is $\mathbf{y} \rightarrowtriangle \mathbf{y}$, the abstraction of $\{y = \mathtt{f}(\mathtt{a})\}$ is $\mathbf{x} \rightarrowtriangle \mathbf{x}$ and the abstraction of $\{x = \mathtt{f}(\mathtt{a}), y = \mathtt{f}(\mathtt{a})\}$ is $\emptyset \rightarrowtriangle \emptyset$.

Considering $V = \{x, y, w, z\}$, the constraint $\{x = \mathtt{f}(y)\}$ belongs to $\mathbf{z} \rightarrowtriangle \mathbf{z}$. Then, from the point of view of internal freeness dependencies, z is free after composition with a constraint where z is free, where composition means application of the internal$^{\mathcal{P}(H_V)}$ operator.

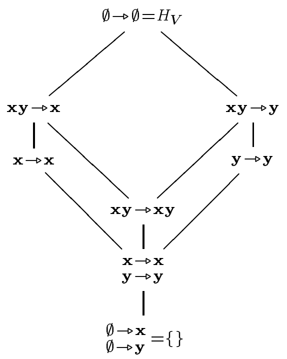

Figure1. The domain $Free^1_{\{x,y\}}$ of internal freeness dependencies

The abstraction map $\alpha_{Free} : \mathcal{P}(H_V) \mapsto Free^1_V$ is induced by the abstract domain $Free^1_V$. Namely, the pair $\langle \alpha_{Free}, \gamma_{Free} \rangle$ is defined as

$$\alpha_{Free}(S) = \cap\{f \in Free^1_V | S \subseteq f\} \qquad \text{for every } S \in \mathcal{P}(H_V)$$
$$\gamma_{Free}(f) = f \qquad \text{for every } f \in Free^1_V,$$

and forms a Galois insertion between $\mathcal{P}(H_V)$ and Free^1_V.

The concrete operations of Section 3 induce corresponding abstract operations on Free^1_V. We will show that the internal freeness dependencies can be used to model the composition (the $\star^{\mathcal{P}(H_V)}$ operation) in a precise way. The insight is that to compute the global dependencies between two constraints we have to exploit our knowledge of their internal dependencies.

The next section introduces a *computer representation* of the elements of Free^1_V. Later we will provide algorithms on this representation for computing the abstract operations induced by the operations of Section 3.

6 A representation for Free^1_V

The obvious choice for the representation of freeness dependencies is to use pairs of sets of variables, with a distinguished element $\bot$. We will write the pair $\langle\{v_1,\ldots,v_n\},\{v'_1,\ldots,v'_m\}\rangle$, $n,m\geq 1$, as $v_1\cdots v_n \Rightarrow v'_1\cdots v'_m$ and will represent the set of constraints $\mathbf{v_1}\cdots\mathbf{v_n} \twoheadrightarrow \mathbf{v'_1}\cdots\mathbf{v'_m}$. Let $\mathit{Rep}^1_V=\mathcal{P}(V\star V)\cup\{\bot\}$. The concretisation function $\gamma_{\mathit{Rep}}:\mathit{Rep}^1_V\mapsto\mathit{Free}^1_V$ is defined as

$$\gamma_{\mathit{Rep}}(\bot)=\{\}\ ,\qquad \gamma_{\mathit{Rep}}(\{A_1,\ldots,A_k\})=\bigcap_{i=1}^{k}\gamma_{\mathit{Rep}}(A_i)\qquad\text{where}$$
$$\gamma_{\mathit{Rep}}(v_1\cdots v_n\Rightarrow v'_1\cdots v'_m)=\mathbf{v_1}\cdots\mathbf{v_n}\twoheadrightarrow\mathbf{v'_1}\cdots\mathbf{v'_m}\ .$$

Note that top is represented by the empty set. Since different elements of Rep^1_V can have the same concretisation, we consider an equivalence relation over Rep^1_V defined as $r_1\equiv r_2$ if and only if $\gamma_{\mathit{Rep}}(r_1)=\gamma_{\mathit{Rep}}(r_2)$. From now on, every element of Rep^1_V will stand for its equivalence class, and Rep^1_V will stand for $\mathit{Rep}^1_V/\equiv$. γ_{Rep} will be applied to equivalence classes and the corresponding abstraction function is defined as follows. Given some $I\subseteq\mathbb{N}$ and, for each $i\in I$, $\mathbf{L_i}\twoheadrightarrow\mathbf{R_i}\in\mathit{Free}^1_V$, then

$$\alpha_{\mathit{Rep}}(\cap_{i\in I}\{\mathbf{L_i}\twoheadrightarrow\mathbf{R_i}\})=\{L_i\Rightarrow R_i \mid i\in I\}\ . \tag{4}$$

It can be easily checked that $\langle\alpha_{\mathit{Rep}},\gamma_{\mathit{Rep}}\rangle$ is a Galois insertion between Free^1_V and Rep^1_V. Every operation in $\mathcal{P}(H_V)$ induces a corresponding operation in Free^1_V which induces a corresponding operation in Rep^1_V. To use Rep^1_V for freeness analysis of logic programs we must provide

1. an algorithm for computing $\alpha_{\mathit{Rep}}\alpha_{\mathit{Free}}(\{h\})$ for every $h\in H_V$ (Section 7);
2. an algorithm for computing the set of variables that are free in every Herbrand constraint belonging to the concretisation of an element of Rep^1_V (Section 7);
3. an algorithm for computing the operation induced by $\star^{\mathcal{P}(H_V)}$ (Section 8);
4. an algorithm for computing the operation induced by $\exists_x^{\mathcal{P}(H_V)}$ for every $x\in V$ (Section 8);
5. an element of Rep^1_V corresponding to $\delta_{\tilde{x},\tilde{y}}^{\mathcal{P}(H_V)}$ for every sequences $\tilde{x}$ and $\tilde{y}$ whose intersection is empty (Section 8).

7 The abstraction function

In this section we provide an algorithmic definition of the restriction of the function $\alpha_{Rep}\alpha_{Free} : \mathcal{P}(H_V) \mapsto Rep^1_V$ to singletons.

Definition 4. *Given $v \in V$, we define $L_c(v) = \{v' \in V | \, \mathrm{vars}(c(v')) \cap \mathrm{vars}(c(v)) \neq \emptyset\}$. We define $\alpha_{alg}(\exists_W c) = \{(L_c(v) \setminus W) \Rightarrow v | v \in \mathrm{free}(\exists_W c)\}$.*

We prove that $\alpha_{alg}(h)$ is equivalent to $\alpha_{Rep}\alpha_{Free}(\{h\})$ for every $h \in H_V$.

Proposition 5. *Given $h \in H_V$, we have $\alpha_{alg}(h) = \alpha_{Rep}\alpha_{Free}(\{h\})$.*

Example 6. Consider $V = \{x, y, v, z\}$. The table below shows some Herbrand constraints and their abstraction through α:

h	$\alpha_{alg}(h)$
$\{v = \mathtt{f}(x), z = \mathtt{a}\}$	$\{vx \Rightarrow x, y \Rightarrow y\}$
$\{v = \mathtt{g}(x, y), z = \mathtt{f}(y)\}$	$\{vx \Rightarrow x, vyz \Rightarrow y\}$
$\{v = \mathtt{g}(x, y), z = y\}$	$\{vx \Rightarrow x, vyz \Rightarrow y, vyz \Rightarrow z\}$
$\{v = x, y = x\}$	$\{vxy \Rightarrow x, vxy \Rightarrow y, vxy \Rightarrow v, z \Rightarrow z\}$
$\exists_{\{w,s\}} \{v = \mathtt{f}(w, y), s = \mathtt{f}(x), z = w\}$	$\{x \Rightarrow x, yv \Rightarrow y, vz \Rightarrow z\}$

Definition 7. *$A \in Rep^1_V$ is in normal form if it is $\perp$ or if $A = \{B_1 \Rightarrow v_1, \ldots, B_n \Rightarrow v_n\}$ with $v_i \in B_i$ for every $i = 1, \ldots, n$ and $v_i \neq v_j$ for every $i, j = 1, \ldots, n$, $i \neq j$.*

Note that, by definition, $\alpha_{alg}(h)$ is in normal form for every $h \in H_V$.

Proposition 8. *For all $A \in Rep^1_V$ there is an $A' \in Rep^1_V$ in normal form such that $\gamma_{Rep}(A) = \gamma_{Rep}(A')$.*

In the following, we always consider elements in normal form as representative of elements of Rep^1_V.

We want an algorithm for computing the set of variables that are free in every consistent Herbrand constraint belonging to the concretisation of $A \in Rep^1_V$. The following definition provides such an algorithm:

Definition 9. *We define the map $\mathrm{free}_{alg} : Free^1_V \mapsto \mathcal{P}(V)$ as $\mathrm{free}_{alg}(\perp) = V$ and $\mathrm{free}_{alg}(\{B_1 \Rightarrow v_1 \cdots B_n \Rightarrow v_n\}) = \{v_1, \ldots, v_n\}$.*

Proposition 10. *Given $A \in Rep^1_V$, we have $\cap_{h \in \gamma_{Free}\gamma_{Rep}(A)} \mathrm{free}(h) = \mathrm{free}_{alg}(A)$.*

8 The abstract operators

In this section we define the abstract counterparts of the conjunction and restriction operators, as well as of the diagonal elements of Section 3.

The abstract conjunction operator constructs an arrow for a variable v by unfolding the body of an arrow for v contained in A_1 with the arrows in A_2 and so on alternately. If this unfolding fails, no arrow is built for the variable v.

Definition 11 (Conjunction on Rep^1_V). *Given $A_1, A_2 \in Rep^1_V$ we define*

$$A_1 \star^{Free^1_V} A_2 = \begin{cases} \perp & \textit{if } A_1 = \perp \textit{ or } A_2 = \perp \\ \left\{ \bigcup_{i \geq 0} \mathrm{dunf}^i_{A_1,A_2}(B) \Rightarrow v \;\middle|\; \begin{array}{l} B \Rightarrow v \in A_1, \\ \mathrm{dunf}^i_{A_1,A_2}(B) \\ \textit{is not fail} \\ \textit{for every } i \geq 0 \end{array} \right\} & \textit{otherwise,} \end{cases}$$

where $\mathrm{dunf}^1_{A_1,A_2}(B) = B$, $\mathrm{dunf}^i_{A_1,A_2}(B) = \mathrm{unf}(B, A_{1+((i-1) \bmod 2)})$ *for* $i \geq 2$, *and*

$$\mathrm{unf}(fail, A) = fail \qquad \mathrm{unf}(\emptyset, A) = \emptyset$$

$$\mathrm{unf}(v_1 \cdots v_n, A) = \begin{cases} B_1 \setminus v_1 \cdots B_n \setminus v_n & \textit{if } \exists (B_i \Rightarrow v_i) \in A \textit{ for } i = 1, \ldots, n \\ fail & \textit{otherwise.} \end{cases}$$

Note that the definition above provides an algorithm for computing $\star^{Free^1_V}$, since, V being finite, there must exist two different natural numbers i, j, both even or both odd, such that $\mathrm{dunf}^i_{A_1,A_2}(B) = \mathrm{dunf}^j_{A_1,A_2}(B)$. Then the computation of the union can be stopped at the $\max(i, j)$-th iteration. Moreover, note that the operation $\star^{Rep^1_V}$ is closed on the set of elements of Rep^1_V that are in normal form.

Example 12. Consider the two sets of arrows $A_1 = \{xy \Rightarrow x, vz \Rightarrow v, vz \Rightarrow z\}$ and $A_2 = \{yz \Rightarrow y, x \Rightarrow x, v \Rightarrow v\}$. Consider the variable x. There is an arrow $xy \Rightarrow x$ for x in A_1. Then we start unfolding xy and we obtain $\mathrm{dunf}^1_{A_1,A_2} = xy$, $\mathrm{dunf}^2_{A_1,A_2} = z$, $\mathrm{dunf}^3_{A_1,A_2} = v$ and $\mathrm{dunf}^i_{A_1,A_2} = \emptyset$ for every $i \geq 4$. Hence the abstract conjunction contains the arrow $vxyz \Rightarrow x$. If we consider the variable y, we do not find any arrow for y in A_1. Then no arrow for y is computed by the algorithm. If we consider the variable v, we have $\mathrm{dunf}^1_{A_1,A_2} = vz$ and $\mathrm{dunf}^2_{A_1,A_2} = fail$, since there is no arrow for z in A_2. Then the algorithm does not add any arrow for v in the abstract unification.

Note that $h_1 = \{y = \mathtt{f}(x), z = v\} \in \gamma_{Free}\gamma_{Rep}(A_1)$ and $h_2 = \{z = \mathtt{g}(y)\} \in \gamma_{Free}\gamma_{Rep}(A_2)$. Their concrete conjunction is $h = \{v = \mathtt{g}(\mathtt{f}(x)), y = \mathtt{f}(x), z = \mathtt{g}(\mathtt{f}(x))\}$. Note that neither y nor v are free in h. Then it is correct that the abstract unification algorithm does not contain any arrow for those two variables. Moreover, note that the arrow $vxyz \Rightarrow x$, computed by the algorithm, is correct for h.

Proposition 13. *Given $A_1, A_2 \in Rep^1_V$ we have*

$$\gamma_{Free}\gamma_{Rep}\left(A_1 \star^{Free^1_V} A_2\right) \supseteq \gamma_{Free}\gamma_{Rep}(A_1) \star^{\mathcal{P}(H_V)} \gamma_{Free}\gamma_{Rep}(A_2) \ .$$

Definition 14 (Restriction on Rep^1_V). *Given $A \in Rep^1_V$ and $x \in V$ we define*

$$\exists_x^{Rep^1_V}(A) = \begin{cases} \{x \Rightarrow x\} \cup \{(L - x) \Rightarrow (R - x) | L \Rightarrow R \in A, R \neq \{x\}\} & \textit{if } A \neq \perp \\ \perp & \textit{otherwise.} \end{cases}$$

Note that the operation $\exists_x^{Rep^1_V}$ is closed on the set of elements of Rep^1_V that are in normal form.

Proposition 15. *Given $A \in Rep^1_V$ and a variable x, we have*

$$\gamma_{Free}\gamma_{Rep}\left(\exists_x^{Rep^1_V} A\right) = \exists_x^{\mathcal{P}(H_V)}\gamma_{Free}\gamma_{Rep}(A) \ .$$

Definition 16 (Diagonal elements on Rep^1_V). *Given $\tilde{x} = \langle x_1, \dots, x_n\rangle$ and $\tilde{y} = \langle y_1, \dots, y_n\rangle$ formed by distinct variables and whose intersection is empty, we define*

$$\delta_{\tilde{x},\tilde{y}}^{Rep^1_V} = \{x_i y_i \Rightarrow y_i, x_i y_i \Rightarrow x_i | i = 1, \dots, n\} \ .$$

Note that $\delta_{\tilde{x},\tilde{y}}^{Rep^1_V}$ is in normal form.

Proposition 17. *Given $\tilde{x} = \langle x_1, \dots, x_n\rangle$ and $\tilde{y} = \langle y_1, \dots, y_n\rangle$ formed by distinct variables and whose intersection is empty, we have*

$$\alpha_{Rep}\alpha_{Free}\left(\delta_{\tilde{x},\tilde{y}}^{\mathcal{P}(H_V)}\right) = \delta_{\tilde{x},\tilde{y}}^{Rep^1_V} \ .$$

9 An example

We show here a simple example of freeness analysis of logic programs, based on a call pattern version of the s-semantics [2]. Given an input goal for a procedure and a selection rule, a (concrete) call pattern is the name of a procedure call found during the resolution of the goal, together with the partial Herbrand constraint computed up to that call, restricted to the arguments of the procedure. Note that the input goal itself is a call pattern. Let us call $\iota_1, \dots, \iota_n$ the arguments of the input goal and $\kappa_1, \dots, \kappa_m$ the arguments of a call pattern found during the resolution of the input goal. A call pattern semantics collects every call pattern found during the resolution of a goal. An abstract call pattern semantics is the same, except that it uses abstract constraints instead of concrete ones. Here, we will use elements of $Free^1_V$ as the abstract constraints and we will assume a leftmost selection rule.

Consider the following program, taken from [20], that computes the product of a matrix and a vector:

```
multiply([],V,[]).
multiply([VM|MM],V,[H|T]):-vmul(VM,V,H),multiply(MM,V,T).

vmul([],[],0).
vmul([H1|T1],[H2|T2],R):-vmul(T1,T2,R1),R is (R1+H1*H2).
```

The call pattern semantics of the predicate `vmul` contains just the following two call patterns, both for `vmul`:

$\{\iota_1\kappa_1 \Rightarrow \kappa_1, \iota_2\kappa_2 \Rightarrow \kappa_2, \iota_3\kappa_3 \Rightarrow \kappa_3, \iota_1\kappa_1 \Rightarrow \iota_1, \iota_2\kappa_2 \Rightarrow \iota_2, \iota_3\kappa_3 \Rightarrow \iota_3\}$, `vmul`
$\{\iota_3 \Rightarrow \iota_3, \iota_1\kappa_1 \Rightarrow \kappa_1, \iota_2\kappa_2 \Rightarrow \kappa_2, \kappa_3 \Rightarrow \kappa_3\}$, `vmul`

The first call pattern corresponds to the same input pattern of the procedure, while the second call pattern corresponds to every recursive call. Both call patterns contain the arrow $\iota_i\kappa_i \Rightarrow \kappa_i$ for $i = 1, 2$. The first contains the arrow $\iota_3\kappa_3 \Rightarrow \kappa_3$ and the second the arrow $\kappa_3 \Rightarrow \kappa_3$. If we make the conjunction ($\star^{Rep^1_V}$) of this denotation with the input pattern $\{\iota_i \Rightarrow \iota_i, \kappa_i \Rightarrow \kappa_i | 1 \leq i \leq 3\}$, the i-th argument of every concrete call pattern (κ_i) is free. Note that the second abstract call pattern says that the third argument of every concrete call pattern for `vmul` (κ_3) is free independently from the input pattern (indeed, it contains the arrow $\kappa_3 \Rightarrow \kappa_3$). This means that in every recursive call found in the concrete resolution, the third argument is always free.

The denotation of `multiply` contains more interesting information. It contains call patterns for `multiply` as well as call patterns for `vmul`:

$\{\iota_1\kappa_1 \Rightarrow \kappa_1, \iota_2\kappa_2 \Rightarrow \kappa_2, \iota_3\kappa_3 \Rightarrow \kappa_3, \iota_1\kappa_1 \Rightarrow \iota_1, \iota_2\kappa_2 \Rightarrow \iota_2, \iota_3\kappa_3 \Rightarrow \iota_3\}$, `multiply`
$\{\iota_1\kappa_1 \Rightarrow \kappa_1, \iota_3\kappa_3 \Rightarrow \kappa_3\}$, `multiply`
$\{\iota_2\kappa_2 \Rightarrow \iota_2, \iota_1\kappa_1 \Rightarrow \kappa_1, \iota_2\kappa_2 \Rightarrow \kappa_2, \iota_3\kappa_3 \Rightarrow \kappa_3\}$, `vmul`
$\{\iota_1\kappa_1 \Rightarrow \kappa_1, \iota_2\kappa_2 \Rightarrow \kappa_2, \kappa_3 \Rightarrow \kappa_3\}$, `vmul`
$\{\iota_1\kappa_1 \Rightarrow \kappa_1, \iota_3\kappa_3 \Rightarrow \kappa_3\}$, `vmul`
$\{\iota_1\kappa_1 \Rightarrow \kappa_1, \kappa_3 \Rightarrow \kappa_3\}$, `vmul`

Both call patterns for `multiply` contain the arrow $\iota_3\kappa_3 \Rightarrow \kappa_3$. The presence of this arrow in the second call pattern for `multiply` means that our domain has been able to capture the freeness dependency of the variable `T` from the third argument of the head of the second clause for `multiply`. If we make the conjunction of the denotation above with an input pattern for `multiply` whose third argument (ι_3) is free ($\{\iota_3 \Rightarrow \iota_3, \kappa_i \Rightarrow \kappa_i | 1 \leq i \leq 3\}$), we conclude that every concrete call pattern for `multiply` will have its third argument (κ_3) free. As noted in [20], if we assume that `multiply` is always called with its third argument free and the first two arguments ground, that information allows the two calls `vmul(VM, V, H)` and `multiply(MM, V, T)` to be executed in AND parallelism [14].

10 Conclusion

This paper provides the first new domain to be constructed solely by use of the linear refinement technique. This domain, which is for freeness analysis, enjoys a natural definition and simple abstract operators. An interesting consequence of our investigation is that the standard approach of linear refinement did not lead to a useful domain. Instead, we found a novel approach which we called *internal dependencies*.

The precision of Free^1_V needs to be compared with alternative domains for freeness analysis, for instance the $\mathit{Sharing}_V \times \mathit{Free}^0_V$ domain. It has already been shown that for just the freeness information, the domain $\mathit{Sharing}_V \times \mathit{Free}^0_V$ contains redundant information and that $(\downarrow\mathit{Sharing}_V) \times \mathit{Free}^0_V$ is just as precise [24]. The domain $\downarrow\mathit{Sharing}_V$ [10] is a subset of $\mathit{Sharing}_V$, formed by those points whose sharing sets are downward closed with respect to set inclusion. It can be shown that the Rep^1_V domain is a subset of the reduced product $(\downarrow Sharing_V) \times Free^0_V$. Therefore, freeness analysis performed with Rep^1_V cannot be more precise than freeness analysis performed with $(\downarrow Sharing_V) \times Free^0_V$ (or $Sharing_V \times Free^0_V$ [24]). Our opinion is that the two domains provide the same information with respect to freeness. This argument still needs to be investigated further, possibly proving it through the quotient technique [7].

Condensing does not hold for our domain. As shown in [22], under suitable conditions a domain is condensing if and only if it is closed with respect to $\multimap^{\star\mathcal{P}(H_V)}$. Our domain is not closed with respect to such a refinement. This is not surprising, since we refined a basic domain for freeness with respect to an operation that is not unification. Note, however, that our domain still yields correct results using a goal-independent semantics. The actual gain in precision when using a goal-dependent semantics is not easily quantifiable without an experimental evaluation.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. Technical report, Monash University, 1995.

[2] 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.

[3] M. Bruynooghe and M. Codish. Freeness, Sharing, Linearity and Correctness – All at Once. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Proceedings of the third International Workshop on Static Analysis*, volume 724 of *Lectures Notes in Computer Science*, pages 153–164. Springer-Verlag, 1993.

[4] M. Codish, D. Dams, G. Filé, and M. Bruynooghe. On the Design of a Correct Freeness Analysis for Logic Programs. *Journal of Logic Programming*, 28(3):181–206, September 1996.

[5] A. Cortesi and G. Filè. Abstract Interpretation of Logic Programs: an Abstract Domain for Groundness, Sharing, Freeness and Compoundness Analysis. In *Proc. ACM Symposium on Partial Evaluation and Semantics-based Program Transformation*, pages 52–61. ACM Press, 1991.

[6] A. Cortesi, G. Filè, and W. Winsborough. Optimal Groundness Analysis Using Propositional Logic. *Journal of Logic Programming*, 27(2):137–167, 1996.

[7] A. Cortesi, G. Filé, and W. Winsborough. The Quotient of an Abstract Interpretation. *Theoretical Computer Science*, 202(1-2):163–192, 1998.

[8] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[9] S.K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, July 1989.

[10] Christian Fecht. An Efficient and Precise Sharing Domain for Logic Programs. In *Proceedings of PLILP'96 Programming Languages: Implementations, Logics, and Programs Conference*, volume 1140 of *Lecture Notes in Computer Science*, pages 469–470. Springer-Verlag, 1996.

[11] G. Filé, R. Giacobazzi, and F. Ranzato. A Unifying View on Abstract Domain Design. *ACM Computing Surveys*, 28(2):333–336, 1996.

[12] R. Giacobazzi, F. Ranzato, and F. Scozzari. Building Complete Abstract Interpretations in a Linear Logic-based Setting. In G. Levi, editor, *Static Analysis, Proceedings of the Fifth International Static Analysis Symposium SAS 98*, volume 1503 of *Lecture Notes in Computer Science*, pages 215–229. Springer-Verlag, 1998.

[13] R. Giacobazzi and F. Scozzari. A Logical Model for Relational Abstract Domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.

[14] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In D. H. D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming*, pages 237–252. The MIT Press, 1990.

[15] A. King and P. Soper. Depth-k Sharing and Freeness. In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, pages 553–568, Santa Margherita Ligure, Italy, 1994.

[16] Anno Langen. *Advanced Techniques for Approximating Variable Aliasing in Logic Programs.* PhD thesis, University of Southern California, 1991.

[17] Giorgio Levi and Fausto Spoto. An Experiment in Domain Refinement: Type Domains and Type Representations for Logic Programs. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Proceedings of the 10th International Symposium on Programming Languages, Implementations, Logics and Programs, PLILP'98*, volume 1490 of *Lecture Notes in Computer Science*, pages 152–169, Pisa, Italy, September 1998. ©Springer-Verlag.

[18] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[19] A. Mulkers, W. Simoens, G. Janssesns, and M. Bruynooghe. On the Practicability of Abstract Equation Systems. In L. Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming*, pages 781–795, Kanagawa, Japan, June 1995. The MIT Press.

[20] 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.

[21] F. Scozzari. Logical Optimality of Groundness Analysis. In P. Van Hentenryck, editor, *Proceedings of International Static Analysis Symposium, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1997.

[22] F. Scozzari. *Domain Theory in Abstract Interpretation: Equations, Completeness and Logic.* PhD thesis, Dipartimento di Matematica, Università di Siena, Italy, 1998.

[23] R. Sundararajan and J. Conery. An Abstract Interpretation Scheme for Groundness, Freeness and Sharing Analysis. In R. Shyamasundar, editor, *Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lectures Notes in Computer Science*, pages 203–216, New Delhi, December 1992. Springer-Verlag.

[24] Enea Zaffanella, Roberto Bagnara, and Patricia Hill. Widening Set-Sharing. In *Principles and Practice of Declarative Programming, Proceedings of the First International Conference*, Lecture Notes in Computer Science, Paris, France, September 1999. Springer-Verlag. To appear.