On the Design of a Correct Freeness Analysis for Logic Programs

Michael Codish* Dennis Dams‡ Gilberto Filé§ Maurice Bruynooghe¶

Abstract

Several proposals for computing *freeness* information for logic programs have been put forward in recent literature. The availability of such information has proven useful in a variety of applications, including parallelization of Prolog programs, optimizations in Prolog compilers, as well as for improving the precision of other analyses. While these proposals have illustrated the importance of such analyses, they lack formal justification. Moreover, several have been found incorrect. This paper introduces a novel domain of *abstract equation systems* describing possible sharing and definite freeness of terms in a system of equations. A simple and intuitive abstract unification algorithm is presented, providing the core of a correct and precise sharing and freeness analysis for logic programs. Our contribution is not only a correct algorithm, but perhaps primarily, the application of a systematic approach in which it is derived by mimicking each step in a suitable concrete unification algorithm. Consequently, the abstract algorithm is intuitive — as it resembles the concrete algorithm. It is amenable to formal justification — as the proof of correctness boils down to showing that each step in the concrete algorithm is mimicked by a corresponding step in the abstract algorithm. Finally, it is precise — as each step mimics only those situations which can arise in the concrete algorithm.

To appear in the Journal of Logic Programming.

This article is a revision of [5] (also see Computing Science Note 93/21 of the Department of Computing Science, Eindhoven University of Technology).

*Dept. of Mathematics and Computer Science, Ben-Gurion University of the Negev, PoB 653 Beer-Sheba 84105, Israel. codish@cs.bgu.ac.il.

‡Dept. of Philosophy, Utrecht University, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands. Dennis.Dams@phil.ruu.nl.

§Dip. di Mat. Pura e App., Universitá di Padova, Italy. file@pdmat1.unipd.it

¶Dept. of Computer Science, KU Leuven, Belgium. maurice@cs.kuleuven.ac.be.

1 Introduction

We say that terms t_1 and t_2 *share* if they contain a common variable; they *share under an equation system* E if the terms $t_1\theta$ and $t_2\theta$ share where θ is a most general unifier of E. We say that variable X is *free* under E if $X\theta$ is a variable. Sharing and freeness information are useful for several purposes, for example, in the context of parallel execution of Prolog programs [13]. Consider a clause

$$p(X,Y) \leftarrow q(X), r(Y).$$

A sharing analysis may enable parallel execution of $q(X)$ and $r(Y)$ if it determines that X and Y do not share under any calling pattern. However, even if X and Y do share, then parallel execution is still possible if a freeness analysis determines that execution of $q(X)$ leaves X free.

Freeness information may also be used to optimize programs containing built-ins such as $var(X)$, $nonvar(X)$ or X *is* $Y + Z$. Moreover it can be used to improve the results of groundness and sharing analyses, as described in [19].

We consider analyses which are given within the semantic based framework of abstract interpretation [9]. A program analysis is viewed as a non-standard semantics defined over a domain of data-descriptions. Analyses are constructed by replacing the basic operations on data in a suitable concrete semantics with corresponding abstract operations defined on data-descriptions. Formal justification is reduced to proving conditions on the relation between data and data-descriptions and on the elementary operations defined on the data-descriptions. This approach eases both the development and the justification of program analyses. In the case of logic programming languages, proving the correctness of an abstract unification function is the major step in justifying an analysis.

In this paper we view substitutions as sets of equations in solved form, and unification as the process of reducing a set of equations to solved form. A goal is a pair $\langle g; E\rangle$ where g is a set of atoms and E is a satisfiable set of equations which specifies an instance of g. A resolution step reduces a goal $\langle \ldots, a, \ldots; E\rangle$ with a (renamed) clause $c = h \leftarrow b_1, \ldots, b_n$ by replacing the atom a by $b_1, \ldots, b_n$ and adding the equation $a = h$ to E if $\{a = h\} \cup E$ is satisfiable. The activated instance of c is specified by $mgu(\{a = h\} \cup E)$. See Figure 1(a).

The core component in developing an abstract interpretation is to design an operation $mgu^{\mathcal{A}}$ which abstracts the unification process, as described in Figure 1(b) where $\mathcal{E}$ is an abstract equation system. The basic correctness condition is that for every E which is described by $\mathcal{E}$, we have that $mgu(\{a = h\} \cup E)$ is described by $mgu^{\mathcal{A}}(\{a = h\} \cup \mathcal{E})$. In the following we focus on specifying an abstract unification algorithm which captures freeness of variables.

Consider a single equation $X = f(A)$ where A is bound to a variable and X is bound to a compound term. Solving this equation obviously may bind A to a compound term. On the other hand, if X is bound to a variable then A will remain free. However, if there is another equation involving X then freeness of A may be affected, through sharing as in the following set of equations:

$$E = \Big\{ X = f(A),\ Y = f(g(B)),\ Y = X \Big\}$$

where $mgu(E)$ binds A to the compound term $g(B)$. This example demonstrates the way freeness is influenced by variable sharing. Sharing between X and Y may be less direct and still affect freeness of A, as in

$$\Big\{ X = f(A), Y = f(g(B)), U = V, h(X,Y) = h(U,V) \Big\}.$$

Figure 1.1: (a) Concrete unification: $mgu(\{a = h\} \cup E)$ specifies the activated instance of the clause. (b) Abstract unification: $mgu^{\mathcal{A}}(\{a = h\} \cup \mathcal{E})$ describes the activated instance of the clause.

Thus, precise inference of freeness will depend on an analysis of variable sharing. In the algorithm we present, the propagation of sharing will in fact constitute a main concern. However, even in the presence of other equations involving X it is sometimes possible to infer freeness of A, like in:

$$E' = \Big\{ X = f(A),\ Y = f(B),\ Y = any\_term(...X...) \Big\}$$

where the third equation contains an arbitrary compound term containing X. Any solved form of E' (if it exists) is of the form:

$$mgu(E') = \Big\{ X = f(A),\ B = some\_term(...A...),\ Y = any\_term(...A...) \Big\}$$

in which A remains a free variable. Our abstract unification algorithm formalizes this reasoning.

Early proposals for freeness analysis in logic programs include [10, 16]. More recent proposals such as [19, 7, 21] aim at improving the precision of the analysis by considering more carefully the effect of sharing information on freeness. Unfortunately, attempts to justify these improved algorithms have so far fallen short (a corrected version of [19] can be found in [12] and a revised version of [21] can be obtained). It is our belief that the general intuition behind these algorithms is correct (as well as the algorithms themselves once "fine tuned"). In fact we adopt the same basic intuition. However, we propose that a more systematic approach in the specification of such algorithms should be taken. More specifically, when attempting to mimic the process of concrete unification for data-descriptions, it is productive to systematically mimic each step in a suitable concrete algorithm. This approach has been illustrated in [4] which provides the first proof of correctness for abstract unification over Søndergaard's domain for sharing analysis described in [20]. This paper applies a similar approach and describes the first systematic derivation of an abstract unification algorithm for freeness analysis.

On the bottom line, this paper contributes a clear and intuitive abstract unification algorithm which is the core component needed to provide a freeness analysis by abstract interpretation. The algorithm is derived and proven safe by mimicking each step in a standard unification algorithm, given a suitable notion of data-descriptions. A main contribution of the paper is in the novel choice of data-descriptions called *abstract equation systems* which fuse concrete and abstract equations. It is this choice which facilitates the application of the methodology adopted from [4]. While this paper focuses primarily on correctness of abstract

unification, the results provide a good foundation for the continuing development of precise and efficient freeness analyses for logic programs [2, 3, 18].

The rest of this paper is organized as follows: Section 2 provides some preliminary definitions and notations. Section 3 introduces our notion of data-descriptions which are systems of abstract equations. Section 4 presents the abstract unification algorithm and provides several examples of its use. Section 5 states the correctness of the abstract unification algorithm. Section 6 relates our abstract domain to the $Share \times Free$ domain of [19], discusses ongoing work which aims to improve both precision and efficiency of analyses, and concludes.

A preliminary version of this paper appeared as [5].

2 Preliminaries

Let Σ be a fixed alphabet of function symbols and Var a denumerable set of variables. We assume a partitioning $\mathcal{P}Var \cup \mathcal{M}Var$ of Var (into infinite sets) so as to distinguish between *program variables* and *meta-variables* which are later introduced to capture possible sharing between terms. The sets of terms constructed from elements of Σ and Var and from Σ and $\mathcal{P}Var$ are denoted respectively $Term$ and $\mathcal{P}Term$.

An *equation* over a set T of terms is an object of the form $t_1 = t_2$ where $t_1, t_2 \in T$. An *equation system* over T is a finite set E of equations over T. The *terms* in an equation system are the terms from the left- and right-hand sides of its equations (i.e., not their subterms), unless stated otherwise. Given an equation system E and an equation e, we let $e :: E$ denote the set $\{e\} \cup E$ with the implicit intention that $e \notin E$. An equation system can be reduced by the classic unification algorithm [17] either to a *solved form* (also called a *most general unifier*) if E is satisfiable, or else to $fail$. The unification algorithm consists of four rewrite rules which are applied to the equations in a system until a solved form is reached (or failure is identified). The algorithm is deterministic in the sense that exactly one rule applies to a given equation; it is confluent in the sense that the solved form is unique (up to renaming). This means that the solved form does not depend on the order in which the equations are considered. The rules, shown in Figure 2.1 provide a terminating algorithm for deriving the solved form under the convention that a rule should only be applied when its application modifies the system of equations. The same convention will be assumed for the rules of the abstract unification algorithm introduced in this paper. Note that this unification algorithm applies the occur check ("$X \notin vars(t)$" in rule 4).

We define a partial function mgu which maps an equation system E to a solved form $mgu(E)$. A reference to $mgu(E)$ implicitly implies that E is satisfiable. The correspondence between equations in solved form and idempotent substitutions is well known (see for example [15]). We say that a variable X is *free* with respect to a (concrete) equation system E if it is free under the substitution corresponding to $mgu(E)$.

We adopt the following conventions: meta-variables and elements of $\mathcal{P}Term$ are respectively denoted by $\mathcal{Q}$, $\mathcal{Z}$, $\mathcal{Z}_1$, etc. and by s, t, s_1, t_1, etc. Sets of variables are typically denoted V, V_1, etc. For any syntactic object s, $vars(s) \subseteq Var$ is the set of all variables occurring in s. The restriction of a substitution θ to a set of variables $V \subseteq Var$ is denoted $\theta|V$.

1. $X = X \quad :: \quad E \quad \overset{\text{remove}}{\rightarrow} \quad E$

2. $f(t_1, \ldots, t_n) = X \quad :: \quad E \quad \overset{\text{switch}}{\rightarrow} \quad X = f(t_1, \ldots, t_n) \quad :: \quad E$

3. $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n) \quad :: \quad E \quad \overset{\text{peel}}{\rightarrow} \quad \{t_i = s_i | i = 1..n\} \cup E$

4. $X = t \quad :: \quad E \quad \overset{\text{subst}}{\rightarrow} \quad X = t \quad :: \quad E[X/t]$ if $X \notin vars(t)$.

Figure 2.1: Concrete Unification.

3 Abstract Equations

The set of equation systems over $\mathcal{P}Term$, denoted Eqs, is referred to as the set of *concrete equation systems*. Concrete equation systems are described by abstract equation systems. These are systems in which certain terms may have been replaced by placeholders called *meta-terms*, thereby loosing information about the exact form of the concrete term being described. However, information about sharing and freeness of terms is maintained as formalised below.

Definition 3.1. meta- and abstract- terms, abstract equations
The set of *meta-terms* is $\mathcal{M}Term = \{\bot[V] | V \subseteq Var\}$. The set of *abstract terms* is $\mathcal{A}Term = Term \cup \mathcal{M}Term$. An *abstract equation system* is an equation system over $\mathcal{A}Term$. The set of abstract equation systems is denoted $\mathcal{A}Eqs$.

Intuitively, each abstract term in an abstract equation system describes a concrete term, as follows. Abstract terms which are *not* meta-terms describe concrete terms that are obtained by applying a substitution which replaces all meta-variables by different fresh program variables. Also meta-terms describe concrete terms, but, in this case, each *occurrence* can represent a different term[1]. In a concrete equation system thus obtained, two concrete terms may only (but do not *have* to) share if the abstract terms describing them share. This restriction is called *coherence* below.

Notice that the set V in a meta-term may contain meta-variables as well as program variables. The symbols τ, ξ, τ_1, ξ_1, etc. are used for occurrences of elements of $\mathcal{A}Term$. We often omit set-brackets in meta-terms and write $\bot[X, Y]$ instead of $\bot[\{X, Y\}]$.

The description relation $\propto$ on $\mathcal{A}Eqs \times Eqs$ is formalized in terms of an *abstract term replacement*.

Definition 3.2. abstract term replacement
An *abstract term replacement* μ is a mapping from (occurrences of) abstract terms to terms such that:

[1] We have chosen not to introduce extra notation to denote *occurrences* of meta-terms because the gain in formality would be outweighted by a strongly decreased readability. Instead, we always clearly indicate when we are dealing with such occurrences.

1. $\mu(X) = X$ for $X \in \mathcal{P}Var$;
2. $\mu(\mathcal{Z}) \in \mathcal{P}Var$ for $\mathcal{Z} \in \mathcal{M}Var$;
3. $\mu(f(\tau_1, \ldots, \tau_n)) = f(\mu(\tau_1), \ldots, \mu(\tau_n))$ for $f(\tau_1, \ldots, \tau_n) \in Term$;
4. $\mu(\bot[V]) \in \mathcal{P}Term$ for an occurrence of $\bot[V] \in \mathcal{M}Term$.

We extend μ to a mapping from $\mathcal{A}Eqs$ to Eqs as follows:

5. $\mu(\tau_1 = \tau_2) \quad = \quad \mu(\tau_1) = \mu(\tau_2)$;
6. $\mu(\varepsilon :: \mathcal{E}) \quad = \quad \mu(\varepsilon) :: \mu(\mathcal{E})$ if ε is not of the form $\bot[V] = \bot[V']$;
7. $\mu(\varepsilon :: \mathcal{E}) \quad = \quad \mu(\varepsilon_1) :: \cdots :: \mu(\varepsilon_n) :: \mu(\mathcal{E})$ if ε is of the form $\bot[V] = \bot[V']$, where $\varepsilon_1, \ldots, \varepsilon_n$ are occurrences of ε (for some $n \geq 0$).

An abstract term replacement μ is said to be *coherent* w.r.t. an equation system $\mathcal{E}$ if for any two (occurrences of) abstract terms or subterms in $\mathcal{E}$, $vars(\mu(\tau_1)) \cap vars(\mu(\tau_2)) \neq \emptyset \;\Rightarrow\; vars(\tau_1) \cap vars(\tau_2) \neq \emptyset$.

From point 7 we see that an abstract equation of the form $\bot[V] = \bot[V']$ may describe any number of concrete equations. This is needed because abstract equation systems do not maintain precise information about the number of concrete equations being described. This stems from the fact that the meta-terms $\bot[V]$ and $\bot[V']$ may describe concrete terms containing arbitrarily many subterms. As a consequence, an operation which mimics the "peeling" of an abstract equation $\bot[V] = \bot[V']$ should give an object which describes arbitrarily many concrete equations. In our abstract unification algorithm, the result of "peeling" the abstract equation $\bot[V] = \bot[V']$ is $\bot[V \cup V'] = \bot[V \cup V']$. This intuition is illustrated by Example 4.2.

Intuitively, the coherence of μ means that it does not introduce sharing in the concrete terms which is not present in the abstract terms. For example, in an abstract equation system $\mathcal{E}$ in which some $X \in \mathcal{P}Var$ occurs, an occurrence $\bot[V]$ of a meta-term may only be mapped to a term which contains X if X is in V (see $\mathcal{E}_1$ in Example 3.1).

Definition 3.3. equation description
An abstract equation system $\mathcal{E}$ *describes* a concrete equation system E, denoted $\mathcal{E} \propto E$, if there exists an abstract term replacement μ, coherent with $\mathcal{E}$, such that $\mu(\mathcal{E}) = E$.

Example 3.1. Consider the following abstract equation systems (letting $A, B, U, W, X, Y \in \mathcal{P}Var$).

$$\mathcal{E}_1 = \left\{ \begin{array}{l} X = f(g(A)) \\ Y = f(B) \\ Y = \bot[X] \end{array} \right\} \qquad \mathcal{E}_2 = \left\{ \begin{array}{l} X = f(A) \\ Y = g(B) \\ X = \bot[\mathcal{Z}] \\ Y = \bot[\mathcal{Z}] \end{array} \right\} \qquad \mathcal{E}_3 = \left\{ \begin{array}{l} X = \mathcal{Z} \\ Y = \bot[\mathcal{Z}] \end{array} \right\}$$

$$\mathcal{E}_4 = \left\{ \bot[\mathcal{Z}] = \bot[\mathcal{Z}] \right\}$$

1. $\mathcal{E}_1 \propto \{X = f(g(A)), Y = f(B), Y = f(f(X))\}$ taking a term replacement μ which maps $\bot[X]$ to $f(f(X))$, because the sharing between X and $f(f(X))$ is present between the corresponding abstract terms (X and $\bot[X]$ resp.) in $\mathcal{E}_1$. Likewise, $\mathcal{E}_1$ describes also

$\{X = f(g(A)), Y = f(B), Y = W\}$ letting μ map $\perp[X]$ to W, because there is only sharing between the two occurrences of Y in E, hence this μ is obviously coherent w.r.t. $\mathcal{E}_1$.

2. $\mathcal{E}_2 \propto \{X = f(A), Y = g(B), X = f(W), Y = g(W)\}$ taking a term replacement μ which maps the two occurrences of $\perp[\mathcal{Z}]$ respectively to $f(W)$ and $g(W)$. The sharing between these terms is allowed because the two meta-term occurrences share $\mathcal{Z}$. However, there does not have to be sharing, as exemplified by the fact that also $\mathcal{E}_2 \propto \{X = f(A), Y = g(B), X = f(W), Y = g(U)\}$.

3. $\mathcal{E}_3 \propto \left\{ X = A, Y = f(A) \right\}$.

4. $\mathcal{E}_4$ describes any concrete system of equations, e.g., $\mathcal{E}_4 \propto \{X = f(A), Y = g(B), X = f(W), Y = g(W)\}$. The motivation for this choice is further clarified in Section 4.

It is sometimes useful to annotate a meta-term $\perp[V]$ denoting that it may be mapped only to a non-variable term, or alternatively, only to a variable. We write $\perp^+[V]$ and $\perp^\bullet[V]$ respectively to denote these cases. The notation $a \equiv b$ will be used to denote that a is of the form b. E.g., we write $\perp[V] \equiv \perp^+[V]$ to specify that $\perp[V]$ is of the form $\perp^+[V]$, and $\perp[V] \not\equiv \perp^\bullet[V]$ to denote that $\perp[V]$ is not of the form $\perp^\bullet[V]$.

The following definition extends the standard notion of *syntactic substitution* for abstract terms. It specifies how to replace all occurrences of a variable in a syntactic (possibly abstract) object by an abstract term τ'.

Definition 3.4. syntactic substitution
Let $\tau, \tau' \in \mathcal{A}Term$ and $X \in Var$. The syntactic substitution of τ' for X in τ is denoted $\tau[X/\tau']$ and is defined as usual for $\tau, \tau' \in Term$. In addition, if τ or τ' is a meta-term then:

$$\tau[X/\tau'] \quad = \quad \begin{cases} \tau & \text{if } X \notin vars(\tau) \\ \perp[vars(\{\tau, \tau'\}) \setminus \{X\}] & \text{otherwise.} \end{cases}$$

Syntactic substitution extends naturally for arbitrary syntactic objects containing abstract terms.

Example 3.2.

1. $\left\{ Y = \perp[X, A], Z = \perp[X, B] \right\} [X/f(a)] \quad = \quad \left\{ Y = \perp[A], Z = \perp[B] \right\}$.

2. $\left\{ X = f(A), Y = \perp[A] \right\} [A/\perp[\mathcal{Z}]] \quad = \quad \left\{ X = \perp[\mathcal{Z}], Y = \perp[\mathcal{Z}] \right\}$.

We note that in general, syntactic substitution does not preserve sharing. The meta-terms $\perp[X, A]$ and $\perp[X, B]$ in Example 3.2(1) describe respectively the terms $g(X, A, Q)$ and $g(X, B, Q)$, because the term replacement μ which maps $\perp[X, A]$ to $g(X, A, Q)$ and $\perp[X, B]$ to $g(X, B, Q)$ is coherent. However, consider the syntactic substitution $[X/f(a)]$:

- $g(X, A, Q)[X/f(a)] = g(f(a), A, Q)$ and $g(X, B, Q)[X/f(a)] = g(f(a), B, Q)$ (observe that $g(f(a), A, Q)$ and $g(f(a), B, Q)$ share the variable Q);

- $\bot[X, A][X/f(a)] = \bot[A]$ and $\bot[X, B][X/f(a)] = \bot[B]$ (observe that $\bot[A]$ and $\bot[B]$ do not share).

Consequently, there is no coherent μ under which $\bot[X, A][X/f(a)]$ and $\bot[X, B]\ [X/f(a)]$ are mapped to $g(X, A, Q)[X/f(a)]$ and $g(X, B, Q)[X/f(a)]$. We would like to have an abstract operation that correctly mimics syntactic substitution, also with respect to sharing. To this end we observe that whenever two meta-terms $\bot[V]$ and $\bot[V']$ in an abstract equation system $\mathcal{E}$ share a variable, it is possible to add a "fresh" meta-variable $\mathcal{Q}$ (i.e., obtaining $\bot[V \cup \{\mathcal{Q}\}]$ and $\bot[V' \cup \{\mathcal{Q}\}]$) without changing the interpretation of $\mathcal{E}$ (i.e., the set of concrete equation systems described by $\mathcal{E}$). Likewise, a meta-variable can always be removed from meta-terms if this does not affect the sharing in $\mathcal{E}$. For a (possibly abstract) syntactic object s, we denote by $s[X + \mathcal{Q}]$ the object obtained by adding the meta-variable $\mathcal{Q}$ in each meta-term containing X in s.

Proposition 3.1 *Let $E \in Eqs$, $\mathcal{E} \in \mathcal{A}Eqs$ and $\mathcal{Q} \in \mathcal{M}Var$. If $\mathcal{Q} \notin vars(\mathcal{E})$ then for every variable X in $\mathcal{E}$, $\quad \mathcal{E} \propto E \quad \Leftrightarrow \quad \mathcal{E}[X + \mathcal{Q}] \propto E$.*

Example 3.3.

1. $\{X = a, Y = \bot[X], Z = \bot[X]\}[X + \mathcal{Q}] \quad = \quad \{X = a, Y = \bot[X, \mathcal{Q}], Z = \bot[X, \mathcal{Q}]\}$.
2. $\mathcal{Q}$ can be removed from $\left\{ X = \bot[\mathcal{Q}, A], Y = \bot[\mathcal{Q}, A, B] \right\}$ but not from $\left\{ X = \bot[\mathcal{Q}, A], Y = \bot[\mathcal{Q}, B] \right\}$ and not from $\left\{ X = \mathcal{Q}, Y = \bot[\mathcal{Q}, B] \right\}$.

Adding and removing variables from meta-terms is used in our abstract unification algorithm. In particular, when performing syntactic substitution we add meta-variables to preserve sharing information and correctly mimic concrete unification. On the other hand, an implementation of the algorithm benefits from the removal of superfluous variables from meta-terms. We assume throughout that meta-terms $\bot[V]$ with $V = \emptyset$ are not allowed. A fresh meta-variable can always be added to such a meta-term and this simplifies our construction.

4 The Abstract Unification Algorithm

The abstract unification algorithm consists of a set of *abstract rewrite rules* which mimic the corresponding rules for concrete unification and are illustrated in Figure 4.1 (where we assume that $\mathcal{Q} \in \mathcal{M}Var$ is a "fresh" meta-variable and $\tau \in Term$). Whenever an abstract equation ε (in an abstract equation system) describes a concrete equation e then there is an abstract rule applicable to ε which corresponds to the concrete rule applicable to e (indicated by a label on the arrow, see Figure 2.1). The algorithm reduces an abstract equation system by repeated application of these rules. Once again, we assume the convention that a rule is applied only if it changes the (abstract) equation system. In this way the rules provide a terminating algorithm. Intuitively, a variable is free if it remains free in *every* sequence of (abstract) rewrites. This notion is formalized in Definition 1.

In contrast to concrete unification, several rules may apply to a given abstract equation, as it may describe different concrete equations. Hence, the abstract unification algorithm is non-deterministic and may result in different solved forms, *all* of which must be considered. The

algorithm is also not confluent. Namely, choosing abstract equations in different orders may result in different (sets of) solved forms. However, correctness is maintained regardless of the order in which equations are considered, as is proven in the next section. The loss of confluence in algorithms of this type is not uncommon (see also [4]). It stems from the fact that different orders of performing a set of (abstract) actions may involve different approximations of data. In particular, we may apply heuristics to choose orders which are more likely to involve less approximation and hence provide more precise results. In examples we adopt the convention that the equation chosen for (abstract or concrete) reduction is indicated by underlining it. The rule that is applied in an abstract reduction is indicated by labelling the arrow by the number of the rule. When we say, in the examples which follow, that an abstract reduction $\mathcal{E} \to \mathcal{E}'$ *correctly mimics* a concrete reduction $E \to E'$, we mean that $\mathcal{E} \propto E$ and $\mathcal{E}' \propto E'$. The formal proof of correctness given in Section 5 basically shows that every reduction in a concrete unification is mimicked by some abstract reduction in a corresponding abstract unification.

The rules in Figure 4.1 are classified according to the form of ε, the abstract equation chosen for reduction. In each case we mimic any concrete rule which might apply for some concrete equation system described by the abstract system. Rules 1–3 are identical to their concrete counterparts; also 5/6(a&b) are easily motivated (note that the condition $\tau \in V$ in these rules implies that τ is a variable). Rules 5(c) and 6(c) are motivated by the possibility that the chosen abstract equation describes an equation of the form $f(\ldots) = f(\ldots)$. The rules 4, 5(d) and 6(d) which mimic syntactic substitution are motivated by the following. In rule 4 a fresh meta-variable is added to preserve possible sharing (see Example 4.1(1)). In rule 5(d), observe that while X may occur in V, it may not occur in the concrete term described by $\bot[V]$. Hence, V is replaced by $V' = V[X/\mathcal{Q}]$ on the right of the arrow (see Example 4.1(2)). In rule 6(d) consider that $\bot[V]$ may describe either a (program) variable occurring in $V \setminus vars(\tau)$ or a "hidden" variable, described by $\mathcal{Q}$, which may occur also in a term described by another meta-term $\bot[V']$, but only if V' contains a variable $Y \in V$ (see Example 4.1(3)). Rule 7 is a special case in which no freeness information can be maintained: any variable in $V_1 \cup V_2$ is potentially non-free (see Example 4.2).

Example 4.1. rules 4, 5(d) and 6(d)

1. $\left\{\underline{X = a}, A = \bot[X], B = \bot[X]\right\} \overset{4}{\to} \left\{X = a, A = \bot[\mathcal{Q}], B = \bot[\mathcal{Q}]\right\}$

   correctly mimics the following concrete transitions:

   $$\left\{\underline{X = a}, A = f(X), B = g(X)\right\} \overset{\text{subst}}{\to} \left\{X = a, A = f(a), B = g(a)\right\},$$

   $$\left\{\underline{X = a}, A = f(X, U), B = g(X, U)\right\} \overset{\text{subst}}{\to} \left\{X = a, A = f(a, U), B = g(a, U)\right\}.$$

2. $$\left\{\begin{array}{l}\underline{Y = X}\\ Y = \bot[X]\\ Z = \bot[X]\end{array}\right\} \overset{4}{\to} \left\{\begin{array}{l}Y = X\\ \underline{X = \bot[X]}\\ \overline{Z = \bot[X]}\end{array}\right\} \overset{5(\text{d})}{\to} \left\{\begin{array}{l}Y = \bot[\mathcal{Q}]\\ X = \bot[\mathcal{Q}]\\ Z = \bot[\mathcal{Q}]\end{array}\right\}$$

Recall that:
$X \in Var$
$\mathcal{Q} \in \mathcal{M}Var$ is a "fresh" meta-variable
$\tau \in Term$

1. $X = X \;\; :: \;\; \mathcal{E} \;\; \stackrel{\text{remove}}{\rightarrow} \;\; \mathcal{E}.$

2. $f(\tau_1, \ldots, \tau_n) = X \;\; :: \;\; \mathcal{E} \;\; \stackrel{\text{switch}}{\rightarrow} \;\; X = f(\tau_1, \ldots, \tau_n) \;\; :: \;\; \mathcal{E}.$

3. $f(\tau_1, \ldots, \tau_n) = f(\xi_1, \ldots, \xi_n) \;\; :: \;\; \mathcal{E} \stackrel{\text{peel}}{\rightarrow} \{\tau_i = \xi_i | i = 1..n\} \cup \mathcal{E}.$

4. $X = \tau \;\; :: \;\; \mathcal{E} \;\; \stackrel{\text{subst}}{\rightarrow} \;\; X = \tau \;\; :: \;\; \mathcal{E}[X + \mathcal{Q}][X/\tau] \qquad \text{if } X \notin vars(\tau).$

5. $\tau = \bot[V] \;\; :: \;\; \mathcal{E}$

(a) $\stackrel{\text{remove}}{\rightarrow} \;\; \mathcal{E} \qquad \text{if } \tau \in V \text{ and } \bot[V] \not\equiv \bot^+[V].$

(b) $\stackrel{\text{switch}}{\rightarrow} \;\; \bot^\bullet[V] = \tau \;\; :: \;\; \mathcal{E} \qquad \text{if } \tau \equiv f(\tau_1, \ldots, \tau_n) \text{ and } \bot[V] \not\equiv \bot^+[V].$

(c) $\stackrel{\text{peel}}{\rightarrow} \;\; \{\tau_i = \bot[V] | i = 1..n\} \;\cup\; \mathcal{E} \qquad \text{if } \tau \equiv f(\tau_1, \ldots, \tau_n) \text{ and } \bot[V] \not\equiv \bot^\bullet[V].$

(d) $\stackrel{\text{subst}}{\rightarrow} \;\; X = \bot[V'] \;\; :: \;\; \mathcal{E}[X + \mathcal{Q}][X/\bot[V']]$, where $V' = V[X/\mathcal{Q}]$
if $\tau \equiv X$.

6. $\bot[V] = \tau \;\; :: \;\; \mathcal{E}$

(a) $\stackrel{\text{remove}}{\rightarrow} \;\; \mathcal{E} \qquad \text{if } \tau \in V \text{ and } \bot[V] \not\equiv \bot^+[V].$

(b) $\stackrel{\text{switch}}{\rightarrow} \;\; \tau = \bot^+[V] \;\; :: \;\; \mathcal{E} \qquad \text{if } \tau \in Var \text{ and } \bot[V] \not\equiv \bot^\bullet[V].$

(c) $\stackrel{\text{peel}}{\rightarrow} \;\; \{\bot[V] = \tau_i | i = 1..n\} \;\cup\; \mathcal{E} \qquad \text{if } \tau \equiv f(\tau_1, \ldots, \tau_n) \text{ and } \bot[V] \not\equiv \bot^\bullet[V].$

(d) $\stackrel{\text{subst}}{\rightarrow} \begin{cases} (i) \; X = \tau \;\; :: \;\; \mathcal{E}[X + \mathcal{Q}][X/\tau] & \text{for each } X \in V \setminus vars(\tau) \\ (ii) \; \mathcal{Q} = \tau \;\; :: \;\; \mathcal{E}[Y + \mathcal{Q}][\mathcal{Q}/\tau] & \text{for each } Y \in V \end{cases}$
if $\bot[V] \not\equiv \bot^+[V]$.

7. $\bot[V_1] = \bot[V_2] \;\; :: \;\; \mathcal{E} \stackrel{\text{all}}{\rightarrow} \bot[V] = \bot[V] \;\; :: \;\; \mathcal{E}[\bigwedge_{X \in V} X/\bot[V]]$

where $V = V_1 \cup V_2$ and $\mathcal{E}[\bigwedge_{X \in V} X/\bot[V]]$ denotes the simultaneous syntactic substitution of all $X \in V$ by $\bot[V]$.

Figure 4.1: Abstract Unification.

correctly mimics

$$\left\{\begin{array}{l}\underline{Y = X}\\ Y = f(U)\\ Z = g(U,X)\end{array}\right\} \overset{\text{subst}}{\rightarrow} \left\{\begin{array}{l}Y = X\\ \underline{X = f(U)}\\ Z = g(U,X)\end{array}\right\} \overset{\text{subst}}{\rightarrow} \left\{\begin{array}{l}Y = f(U)\\ X = f(U)\\ Z = g(U,f(U))\end{array}\right\}.$$

3.
$$\left\{\begin{array}{l}A = f(X), \underline{\perp[A] = f(Y)}\\ \perp[A] = g(W,Z)\end{array}\right\} \begin{array}{l}\overset{6(\mathrm{d})(i)}{\nearrow} \left\{\begin{array}{l}f(Y) = f(X), A = f(Y)\\ \perp[Y,\mathcal{Q}] = g(W,Z)\end{array}\right\}\\ \underset{6(\mathrm{d})(ii)}{\searrow} \left\{\begin{array}{l}A = f(X), \mathcal{Q} = f(Y)\\ \perp[A,Y] = g(W,Z)\end{array}\right\}\end{array}$$

mimic respectively the following concrete steps:

$$\left\{\begin{array}{l}A = f(X), \underline{A = f(Y)}\\ g(U,A) = g(W,Z)\end{array}\right\} \overset{\text{subst}}{\rightarrow} \left\{\begin{array}{l}f(Y) = f(X), A = f(Y)\\ g(U,f(Y)) = g(W,Z)\end{array}\right\},$$

$$\left\{\begin{array}{l}A = f(X), \underline{U = f(Y)}\\ g(U,A) = g(W,Z)\end{array}\right\} \overset{\text{subst}}{\rightarrow} \left\{\begin{array}{l}A = f(X), U = f(Y)\\ g(f(Y),A) = g(W,Z)\end{array}\right\}.$$

Example 4.2. rule 7
Consider an abstract equation $\perp[A] = \perp[B]$ which describes any number of concrete equations of the form $s = t$ in which s possibly shares with A and t with B. For instance, $s = f(W, g(W), A)$ and $t = f(U, B, g(U))$. Observe that

$$\left\{\begin{array}{l}\underline{\perp[A] = \perp[B]}\\ A = C\end{array}\right\} \overset{7}{\rightarrow} \left\{\begin{array}{l}\underline{\perp[A,B] = \perp[A,B]}\\ \perp[A,B] = C\end{array}\right\} \overset{7}{\rightarrow} \left\{\begin{array}{l}\perp[A,B] = \perp[A,B]\\ \perp[A,B] = C\end{array}\right\}$$

correctly mimics the following concrete reductions:

$$\left\{\begin{array}{l}\underline{f(W,g(W),A) = f(U,B,g(U))}\\ A = C\end{array}\right\} \overset{\text{peel}}{\rightarrow} \left\{\begin{array}{l}W = U\\ g(W) = B\\ \underline{A = g(U)}\\ A = C\end{array}\right\} \overset{\text{subst}}{\rightarrow} \left\{\begin{array}{l}W = U\\ g(W) = B\\ A = g(U)\\ g(U) = C\end{array}\right\}.$$

To see that $\{\perp[A,B] = \perp[A,B]\}$ describes $\{W = U, g(W) = B, A = g(U)\}$, consider an abstract term replacement μ which maps six occurrences of the meta-term $\perp[A,B]$ respectively to the six terms W, U, $g(W)$, B, A and $g(U)$. This mapping is coherent hence providing the required result. As the example illustrates, $\perp[V_1] = \perp[V_2]$ needs to be able to represent any number of concrete equations because "peeling" may replace the represented equation by an unknown number of new ones.

The abstract unification algorithm illustrated in Figure 4.1 is derived by considering for each possible form that an abstract equation may take, the set of concrete equations it describes and the set of concrete transitions which should be mimicked. However, some of the

1′. $X = X \;\; :: \;\; \mathcal{E}$

(a) $\overset{\text{remove}}{\rightarrow} \; \mathcal{E}$ if X occurs in $\mathcal{E}$.

(b) $\overset{\text{remove free}}{\rightarrow} \; X = \perp^{\bullet}[\mathcal{Q}] \;\; :: \;\; \mathcal{E}$ if X does not occur in $\mathcal{E}$.

5′. $\tau = \perp[V] \;\; :: \;\; \mathcal{E}$

(a) $\overset{\text{remove}}{\rightarrow} \; \mathcal{E}$ if $\tau \in V$ and $\perp[V] \not\equiv \perp^{+}[V]$.

(b) $\overset{\text{switch \& subst}}{\rightarrow} \begin{cases} (i) \; X = \tau \;\; :: \;\; \mathcal{E}[X + \mathcal{Q}][X/\tau] & \text{for each } X \in V \setminus vars(\tau) \\ (ii) \; \mathcal{Q} = \tau \;\; :: \;\; \mathcal{E}[Y + \mathcal{Q}][\mathcal{Q}/\tau] & \text{for each } Y \in V. \end{cases}$
if $\tau \equiv f(\tau_1, \ldots, \tau_n)$ and $\perp[V] \not\equiv \perp^{+}[V]$

(c) $\overset{\text{peel}}{\rightarrow} \; \{\tau_i = \perp[V] | i = 1..n\} \; \cup \; \mathcal{E}$ if $\tau \equiv f(\tau_1, \ldots, \tau_n)$ and $\perp[V] \not\equiv \perp^{\bullet}[V]$.

(d) $\overset{\text{subst}}{\rightarrow} \; X = \perp[V'] \;\; :: \;\; \mathcal{E}[X + \mathcal{Q}][X/\perp[V']]$, where $V' = V[X/\mathcal{Q}]$
if $\tau \equiv X$.

6′. $\perp[V] = \tau \;\; :: \;\; \mathcal{E} \overset{\text{switch}}{\rightarrow} \tau = \perp[V] \;\; :: \;\; \mathcal{E}$ if $\tau \notin \mathcal{M}Term$.

Figure 4.2: Improved rules 1, 5 and 6.

---

cases are superfluous. Rules 5 and 6 consider symmetric cases and rule 6(d) can always be applied after rule 5(b). Figure 4.2 shows a more concise version of rules 5 and 6. Rule 6′ always switches the sides of an equation and rule 5′(b) combines the switch & substitute. Rule 1′ is a more precise version of the remove rule (1). It captures the case in which the last occurrence of a variable X is being removed and remembers that this variable is free by adding an equation of the form $X = \perp^{\bullet}[\mathcal{Q}]$, with $\mathcal{Q}$ a fresh meta-variable.

We now present several examples of abstract unification. Superfluous variables in meta-terms are not indicated.

Example 4.3. The following analysis determines that A and B remain free for all equation systems described by the initial abstract equation system.

$$\left\{\begin{array}{l} \underline{X = f(A,B)} \\ Y = C \\ Y = \perp[X] \end{array}\right\} \overset{4}{\rightarrow} \left\{\begin{array}{l} X = f(A,B) \\ Y = C \\ \underline{Y = \perp[A,B]} \end{array}\right\} \overset{5'(\mathrm{d})}{\rightarrow} \left\{\begin{array}{l} X = f(A,B) \\ \underline{\perp[A,B] = C} \\ \underline{Y = \perp[A,B]} \end{array}\right\} \overset{6'}{\rightarrow} \left\{\begin{array}{l} X = f(A,B) \\ C = \perp[A,B] \\ Y = \perp[A,B] \end{array}\right\}$$

Example 4.4. The following is a segment of an analysis which determines that there may be an equation system described by the initial abstract equation system for which no variables

remain free.

$$\left\{\begin{array}{l}\underline{X = f(A,B)}\\ X = \bot[Y]\\ Y = C\end{array}\right\} \overset{4}{\to} \left\{\begin{array}{l}X = f(A,B)\\ \underline{f(A,B) = \bot[Y]}\\ Y = C\end{array}\right\} \begin{array}{ll} \overset{5'(c)}{\nearrow} & \left\{\begin{array}{l}X = f(A,B)\\ A = \bot[Y]\\ B = \bot[Y]\\ Y = C\end{array}\right\} \to \cdots \\ \overset{5'(b)(i)}{\longrightarrow} & \left\{\begin{array}{l}X = f(A,B)\\ Y = f(A,B)\\ f(A,B) = C\end{array}\right\} \to \cdots \\ \overset{5'(b)(ii)}{\searrow} & \left\{\begin{array}{l}X = f(A,B)\\ \mathcal{Q} = f(A,B)\\ Y = C\end{array}\right\} \to \cdots \end{array}$$

Example 4.5. The following is a segment of an analysis which determines that the variable A remains free for all equation systems described by the initial abstract equation system.

$$\left\{\begin{array}{l}X = f(A)\\ \underline{Y = f(B)}\\ \overline{Y = \bot[X]}\end{array}\right\} \overset{4}{\to} \left\{\begin{array}{l}X = f(A)\\ Y = f(B)\\ \underline{f(B) = \bot[X]}\end{array}\right\} \begin{array}{ll} \overset{5'(c)}{\nearrow} & \left\{\begin{array}{l}\underline{X = f(A)}\\ Y = f(B)\\ B = \bot[X]\end{array}\right\} \overset{4}{\to} \left\{\begin{array}{l}X = f(A)\\ Y = f(B)\\ B = \bot[A]\end{array}\right\} \to \cdots \\ \overset{5'(b)(i)}{\longrightarrow} & \left\{\begin{array}{l}f(B) = f(A)\\ Y = f(B)\\ X = f(B)\end{array}\right\} \to \cdots \\ \overset{5'(b)(ii)}{\searrow} & \left\{\begin{array}{l}X = f(A)\\ Y = f(B)\\ \mathcal{Q} = f(B)\end{array}\right\} \end{array}$$

5 Correctness of Abstract Unification

Proving correctness of abstract unification boils down to showing that whenever an abstract equation system $\mathcal{E}$ describes a concrete system E, then for every step that E can make in the concrete unification algorithm, there is a corresponding abstract step that $\mathcal{E}$ can make such that the resulting abstract equation system will describe the corresponding resulting concrete equation system.

Proofs can be found in the appendix.

Lemma 5.1 *If $E \to E' \neq fail$ and $\mathcal{E} \propto E$, then there exists $\mathcal{E}'$ such that $\mathcal{E} \to \mathcal{E}'$ and $\mathcal{E}' \propto E'$.*

Moreover, we prove that freeness can be determined by considering all *solved forms* of the abstract unification algorithm. Intuitively, a solved form is an abstract equation system which is invariant under all applicable rules.

Definition 5.1. abstract solved form and abstract freeness

An abstract term is *compound* if it is not a variable and not of the form $\bot^\bullet[V]$. We say that $\mathcal{E} \in \mathcal{AEqs}$ is in *solved form* if the following conditions hold:

1. $\mathcal{E}$ does not contain (inconsistent) equations of the form $f(\tau_1, \ldots, \tau_n) = g(\xi_1, \ldots, \xi_m)$ $(f/n \neq g/m)$ nor of the form $X = \tau$ such that $X \in vars(\tau)$ and $\tau \equiv f(\tau_1, \ldots, \tau_n)$, and

2. reduction with any applicable rule in the abstract unification algorithm does not change $\mathcal{E}$.

Let $\mathcal{E} \in \mathcal{AEqs}$ be in solved form. We say that X is *free* in $\mathcal{E}$ if $X \in vars(\mathcal{E})$ and no equation in $\mathcal{E}$ is of the form $X = \tau$ where $\tau \in \mathcal{ATerm}$ is compound or of the form $\bot[V] = \bot[V]$ where $X \in V$.

First, we establish termination of the abstract unification algorithm. In this theorem, the non-deterministic abstract unification algorithm is alternatively viewed as operating on a *set* of abstract equation systems, every time applying all possible steps (but only those that change the system they are applied to) to the selected equation, until the result contains solved forms only.

Theorem 5.1 *termination of abstract unification*
For every $\mathcal{E}$, applying only steps which do not leave the system invariant, the abstract unification algorithm reaches all solved forms in a finite number of steps.

Theorem 5.2 *correctness of abstract unification*
If $\mathcal{E} \propto E$, $mgu(E) \neq fail$ and $\mathcal{E}$ has solved forms $\mathcal{E}_1, \ldots, \mathcal{E}_n$, then, for some $0 \leq i \leq n$, $\mathcal{E}_i \propto mgu(E)$.

The following theorem states the main correctness result of our freeness analysis.

Theorem 5.3 *correctness of freeness analysis*
If $\mathcal{E} \propto E$ and $\mathcal{E}$ has solved forms $\mathcal{E}_1, \ldots, \mathcal{E}_n$ and X is free in each of $\mathcal{E}_1, \ldots, \mathcal{E}_n$, then X is free in E.

We have proven the abstract algorithm correct for any order in which the abstract equations are selected. However, some orders may yield more precise results, as illustrated by the following example.

Example 5.1. precision I
The following two abstract unifications differ in the order in which the equations are chosen for reduction.

$$1.\ \left\{ \begin{array}{l} \underline{X = f(Y)} \\ \bot[X] = f(Z) \end{array} \right\} \overset{4}{\rightarrow} \left\{ \begin{array}{l} X = f(Y) \\ \underline{\bot[Y] = f(Z)} \end{array} \right\} \overset{6'}{\rightarrow} \left\{ \begin{array}{l} X = f(Y) \\ \underline{f(Z) = \bot[Y]} \end{array} \right\} \begin{array}{l} \overset{5'(\mathrm{b})(i)}{\nearrow} \\ \overset{5'(\mathrm{b})(ii)}{\longrightarrow} \\ \overset{5'(\mathrm{c})}{\searrow} \end{array} \begin{array}{l} \left\{ \begin{array}{l} X = f(f(Z)) \\ Y = f(Z) \end{array} \right\} \\ \left\{ \begin{array}{l} X = f(Y) \\ \mathcal{Q} = f(Z) \end{array} \right\} \\ \left\{ \begin{array}{l} X = f(Y) \\ Z = \bot[Y] \end{array} \right\} \end{array}$$

$$
2.\ \left\{\begin{array}{l} X = f(Y) \\ \underline{\bot[X] = f(Z)} \end{array}\right\} \overset{6'}{\rightarrow} \left\{\begin{array}{l} X = f(Y) \\ \underline{f(Z) = \bot[X]} \end{array}\right\} \begin{array}{l} \overset{5'(\mathrm{b})(i)}{\nearrow} \\ \overset{5'(\mathrm{b})(ii)}{\longrightarrow} \\ \overset{5'(\mathrm{c})}{\searrow} \end{array} \begin{array}{l} \left\{\begin{array}{l} \underline{f(Z) = f(Y)} \\ X = f(Z) \end{array}\right\} \overset{3\ \&\ 4}{\rightarrow} \left\{\begin{array}{l} Z = Y \\ X = f(Y) \end{array}\right\} \\ \left\{\begin{array}{l} X = f(Y) \\ \mathcal{Q} = f(Z) \end{array}\right\} \\ \left\{\begin{array}{l} \underline{X = f(Y)} \\ Z = \bot[X] \end{array}\right\} \overset{4}{\rightarrow} \left\{\begin{array}{l} X = f(Y) \\ Z = \bot[Y] \end{array}\right\} \end{array}
$$

Initially selecting the first equation in (1) results in a solved form (the upper one) which indicates possible non-freeness of Y. However, all solved forms obtained when selecting first the second equation, as in (2), indicate that Y is free. Hence, we can be sure that Y is free in the solution of any equation system described by the initial abstract equation system.

Example 5.1 illustrates that substituting a variable in a meta-term by a compound term introduces imprecision as the structure of the compound is lost. In (2), first a case analysis on the meta-term is performed, and the loss of precision due to the substitution of X by the compound term is avoided. This suggests a strategy where substitutions of variables by compound terms in meta-terms is delayed as much as possible. Note that rule $5'(\mathrm{b})(i)$ in (2) applies a substitution [X/f(Z)]. However, X does not occur in any meta-term to which this substitution is applied.

A related issue which affects precision is illustrated by the following example.

Example 5.2. precision II
Consider the abstract equation system

$$\mathcal{E} = \left\{ f(W) = \bot^{\bullet}[\mathcal{Z}], f(f(U)) = \bot^{\bullet}[\mathcal{Z}] \right\}$$

in which the occurrences of the meta-term $\bot^{\bullet}[\mathcal{Z}]$ correspond to variables which possibly share. This means that either they do share and $\mathcal{E}$ describes a concrete system of the form

$$\left\{ f(W) = A, f(f(U)) = A \right\};$$

or they do not share and the system described is of the form

$$\left\{ f(W) = A, f(f(U)) = B \right\}.$$

In both cases U remains free. Our algorithm will not detect this because choosing either equation involves the substitution of a compound term into a meta-term. For instance,

$$\left\{\begin{array}{l} f(W) = \bot^{\bullet}[\mathcal{Z}] \\ \underline{f(f(U)) = \bot^{\bullet}[\mathcal{Z}]} \end{array}\right\} \overset{5'(\mathrm{b})(i)}{\rightarrow} \left\{\begin{array}{l} \underline{f(W) = \bot[U]} \\ \mathcal{Z} = f(f(U)) \end{array}\right\} \overset{5'(\mathrm{b})(i)}{\rightarrow} \left\{\begin{array}{l} U = f(W) \\ \mathcal{Z} = f(f(f(W))) \end{array}\right\}.$$

Observe that we cannot annotate the meta-term $\bot[U]$ as compound (because the $\bot^{\bullet}[\mathcal{Z}]$ in the first equation may correspond to a variable which does not share with the other occurrence of $\bot^{\bullet}[\mathcal{Z}]$) and hence we must consider the application of rule $5'(\mathrm{b})$ which, as illustrated, indicates that U is possibly non-free. However, note that the abstract equation system

$$\mathcal{E}' = \left\{ f(W) = \mathcal{Z}, f(f(U)) = \bot^{\bullet}[\mathcal{Z}] \right\}$$

is equivalent to $\mathcal{E}$ in the sense that both systems describe the same set of concrete systems. Applying our algorithm to $\mathcal{E}'$ preserves the freeness of U as illustrated by:

$$\left\{\begin{array}{l} f(W) = \mathcal{Z} \\ \underline{f(f(U)) = \perp^{\bullet}[\mathcal{Z}]} \end{array}\right\} \begin{array}{l} \overset{5'(\mathrm{b})(i)}{\nearrow} \\ \\ \underset{5'(\mathrm{b})(ii)}{\searrow} \end{array} \begin{array}{l} \left\{\begin{array}{l} \underline{f(W) = f(f(U))} \\ \mathcal{Z} = f(f(U)) \end{array}\right\} \overset{3}{\rightarrow} \left\{\begin{array}{l} W = f(U) \\ \mathcal{Z} = f(f(U)) \end{array}\right\}. \\ \\ \left\{\begin{array}{l} \underline{f(W) = \mathcal{Z}} \\ \mathcal{Q} = f(f(U)) \end{array}\right\} \overset{2}{\rightarrow} \left\{\begin{array}{l} \mathcal{Z} = f(W) \\ \mathcal{Q} = f(f(U)) \end{array}\right\} \end{array}$$

This example indicates a preferable way to describe possible sharing between free variables. Thus, if an occurrence of a meta-term of the form $\perp^{\bullet}[\mathcal{Z}]$ in an abstract equation system $\mathcal{E}$ only shares with other meta-terms in $\mathcal{E}$ then we should replace that occurrence by $\mathcal{Z}$ (note however that several such replacements may not be performed simultaneously).

6 Discussion

This section introduces an extended notion of equation description which enables a richer domain of application and enables us to relate the expressive power of our domain with that of the domain $Share \times Free$ described in [19]. This indicates that in addition to facilitating the systematic approach of [4], abstract equation systems provide also the basis for a reasonable domain with which to infer freeness information [18]. Finally we describe ongoing work and conclude.

An Extended Notion of Description

The notion of equation description is extended so that "equivalent" equation systems have the same description. With the current definition, the abstract equation system $\mathcal{E} = \{X = \perp[\mathcal{Z}], Y = \perp^{\bullet}[\mathcal{Z}]\}$ describes the concrete equation system $\theta_1 = \{X = f(A), Y = A\}$ but not the equation system $\theta_2 = \{X = f(Y)\}$ which is equivalent with respect to the variables X and Y. Consequently, if $\mathcal{E}$ is intended to describe the initial state of a predicate $p(X,Y)$ then the corresponding freeness analysis is correct for $p(X,Y)\theta_1$ but not necessarily for $p(X,Y)\theta_2$ in spite of the fact that the two atoms are equal up to renaming.

Definition 6.1. equivalence of equation systems
Two concrete systems of equations E_1 and E_2 are said to be *equivalent with respect to a set of variables* V, denoted $E_1 \approx_V E_2$, if there exist most general unifiers θ_1 and θ_2 of E_1 and E_2 such that $\theta_1{\restriction}V = \theta_2{\restriction}V$. The relation $\approx_{\mathcal{P}Var}$ is abbreviated by $\approx$.

Clearly, if $E_1 \approx_V E_2$ then E_1 and E_2 exhibit the same freeness for variables in V. Hence, if $\mathcal{E}$ describes E_1 then the abstract unification of $\mathcal{E}$ provides a safe approximation of the freeness of variables from V in $mgu(E_2)$. Hence:

Definition 6.2. extended equation description
Let $V \subseteq \mathcal{P}Var$ and $E \in Eqs$. We say that $\mathcal{E} \in \mathcal{A}Eqs$ *describes* E *with respect to* V, denoted $\mathcal{E} \propto_V^{ext} E$ iff there exists E' such that $E \approx_V E'$ and $\mathcal{E} \propto E'$. The relation $\propto_{\mathcal{P}Var}^{ext}$ is abbreviated by $\propto^{ext}$.

Example 6.1. We have $\left\{ X = \bot[\mathcal{Z}], Y = \mathcal{Z} \right\} \propto^{ext}_{\{X,Y\}} \left\{ X = f(Y) \right\}$ because $\left\{ X = \quad f(Y) \right\}$ $\approx_{\{X,Y\}} \left\{ X = f(A), Y = A \right\}$ and $\left\{ X = \bot[\mathcal{Z}], Y = \mathcal{Z} \right\} \propto \left\{ X = f(A), Y = A \right\}$.

This extension does not change the abstract unification algorithm in any way. It only extends the class of concrete unifications which are mimicked by a given abstract unification. In particular, it enables us to relate the domain of abstract equation systems with the popular domain $Share \times Free$.

The Abstract Domain $Share \times Free$

One of the widely used domains for freeness analysis of logic programs is the domain $Share \times Free$ introduced in [19] and used in the analyses described in [7],[12] and [21]. We illustrate how an element of the domain $Share \times Free$ can be expressed by an element of our domain. An abstract substitution $\Delta \in Share \times Free$ over a set of variables $V \subseteq \mathcal{P}Var$ is a subset of $\wp(V)$ in which each variable is annotated by fr or nf indicating that it is *definitely free* or *possibly non-free*. Intuitively, each set $S \in \Delta$ represents the fact that the terms to which the variables in S are bound may share one or more variables. If a variable X appears only in a singleton set, then the terms to which it is bound may contain only variables which do not appear in any other term. If a variable X does not occur in any set, then there is no variable that may occur in the terms to which it is bound and thus those terms are definitely ground. For a formal definition of the domains $Share$ and $Share \times Free$ see [14] and [19] respectively.

Let $\Delta = \left\{ S_1, \ldots, S_m \right\}$ be an abstract substitution over V in the domain $Share \times Free$. The translation of Δ to our domain is as follows:

1. associate a distinct meta-variable $\mathcal{Z}_j$ with each set S_j $(j = 1..m)$;
2. for every X in V define the set of meta-variables $V_X = \{\mathcal{Z}_j | X \in S_j\}$;
3. define

$$\begin{aligned} \mathcal{E}_\Delta \quad = \quad & \left\{ X = \bot^{\bullet}[V_X] \,\middle|\, X^{fr} \in \cup\Delta \right\} \cup \\ & \left\{ X = \bot[V_X] \,\middle|\, X^{nf} \in \cup\Delta \right\} \cup \\ & \left\{ X = \bot^{+}[\mathcal{Q}_i] \,\middle|\, X \notin \cup\Delta \right\}, \end{aligned}$$

   where all $\mathcal{Q}_i$'s are fresh variables[2].

Observe that $\mathcal{E}_\Delta$ can often be refined as suggested in Example 5.2 above.

Example 6.2. Let $\Delta = \left\{ \{X^{fr}, Y^{fr}\}, \{Y^{fr}, Z^{fr}\}, \{X^{fr}\}, \{Z^{fr}\} \right\}$ and $E = \{X = a, Y = B, Z = C\}$. So, Δ describes substitutions which map either X and Y or Y and Z to the same

[2] For each program variable X which is definitely ground, a fresh variable $\mathcal{Q}_i$ is introduced and an equation $X = \bot^{+}[\mathcal{Q}_i]$. This could be avoided by introducing a special meta-term $\bot^{g}$ which can describe any ground term.

variable, but not X and Z. Consequently, both Z and C remain free under a solution of any $E \cup E'$ where E' is described by Δ. The translation of Δ to abstract equations is

$$\mathcal{E}_\Delta = \left\{ X = \perp^\bullet[\mathcal{Z}_1, \mathcal{Z}_3], Y = \perp^\bullet[\mathcal{Z}_1, \mathcal{Z}_2], Z = \perp^\bullet[\mathcal{Z}_2, \mathcal{Z}_4] \right\}.$$

After removing the superfluous meta-variables ($\mathcal{Z}_3$, $\mathcal{Z}_4$) and following the refinement suggested in Example 5.2 we obtain:

$$\mathcal{E}'_\Delta = \left\{ X = \mathcal{Z}_1, Y = \perp^\bullet[\mathcal{Z}_1, \mathcal{Z}_2], Z = \mathcal{Z}_2 \right\}.$$

The analysis proceeds applying abstract unification to $E \cup \mathcal{E}'_\Delta$:

$$\left\{ \begin{array}{l} \underline{X = a} \\ \underline{Y = B} \\ \underline{Z = C} \\ X = \mathcal{Z}_1 \\ Y = \perp^\bullet[\mathcal{Z}_1, \mathcal{Z}_2] \\ Z = \mathcal{Z}_2 \end{array} \right\} \overset{3\times\ 4}{\rightarrow} \left\{ \begin{array}{l} X = a \\ Y = B \\ Z = C \\ a = \mathcal{Z}_1 \\ \underline{B = \perp^\bullet[\mathcal{Z}_1, \mathcal{Z}_2]} \\ C = \mathcal{Z}_2 \end{array} \right\} \overset{5'(\mathrm{d})}{\rightarrow} \left\{ \begin{array}{l} X = a \\ Y = \perp^\bullet[\mathcal{Z}_1, \mathcal{Z}_2] \\ Z = C \\ a = \mathcal{Z}_1 \\ B = \perp^\bullet[\mathcal{Z}_1, \mathcal{Z}_2] \\ C = \mathcal{Z}_2 \end{array} \right\} \rightarrow \cdots \rightarrow \left\{ \begin{array}{l} X = a \\ Y = \perp[\mathcal{Z}_2] \\ Z = \mathcal{Z}_2 \\ \mathcal{Z}_1 = a \\ B = \perp[\mathcal{Z}_2] \\ C = \mathcal{Z}_2 \end{array} \right\}$$

indicating as desired that Z and C are definitely free.

The following example highlights a point at which our abstract unification algorithm is bound to be more precise than previous proposals. The example illustrates a situation which may arise in the course of an analysis.

Example 6.3. precision III
Assume two variables X and Y which are definitely free and possibly share, and consider a unification which binds both X and Y to compound terms, for instance:

$$E = \left\{ \begin{array}{l} X = f(W) \\ Y = f(f(U)) \end{array} \right\}$$

The abstract substitution $\Delta = \left\{ \{X^{fr}, Y^{fr}\}, \{X^{fr}\}, \{Y^{fr}\} \right\}$ in $Share \times Free$ captures the sharing and freeness information specified above. Translation into abstract equations and application of the refinement as suggested in Example 5.2 gives (after removing superfluous meta-variables):

$$\mathcal{E}'_\Delta = \left\{ \begin{array}{l} X = \mathcal{Z} \\ Y = \perp^\bullet[\mathcal{Z}] \end{array} \right\}.$$

Solving $E \cup \mathcal{E}'_\Delta$ proceeds as follows:

$$\left\{ \begin{array}{l} \underline{X = f(W)} \\ \underline{Y = f(f(U))} \\ X = \mathcal{Z} \\ Y = \perp^\bullet[\mathcal{Z}] \end{array} \right\} \overset{4\ \times\ 2}{\rightarrow}$$

$$\left\{\begin{array}{l} X = f(W) \\ Y = f(f(U)) \\ f(W) = \mathcal{Z} \\ \underline{f(f(U)) = \perp^{\bullet}[\mathcal{Z}]} \end{array}\right\} \begin{array}{l} \overset{5'(\mathrm{b})(i)}{\nearrow} \\ \\ \underset{5'(\mathrm{b})(ii)}{\searrow} \end{array} \begin{array}{l} \left\{\begin{array}{l} X = f(W) \\ Y = f(f(U)) \\ \underline{f(W) = f(f(U))} \\ \mathcal{Z} = f(f(U)) \end{array}\right\} \rightarrow \cdots \rightarrow \left\{\begin{array}{l} X = f(f(U)) \\ Y = f(f(U)) \\ W = f(U) \\ \mathcal{Z} = f(f(U)) \end{array}\right\} \\ \\ \left\{\begin{array}{l} X = f(W) \\ Y = f(f(U)) \\ \underline{f(W) = \mathcal{Z}} \\ \mathcal{Q} = f(f(U)) \end{array}\right\} \overset{2}{\rightarrow} \left\{\begin{array}{l} X = f(W) \\ Y = f(f(U)) \\ \mathcal{Z} = f(W) \\ \mathcal{Q} = f(f(U)) \end{array}\right\}. \end{array}$$

indicating that U is definitely free. In contrast, the algorithms described in [19, 7, 21] as well as those described in [2, 3, 18] do not capture freeness information for this type of example. It is the nondeterministic approach which enables our algorithm to distinguish between two cases and to maintain freeness information in each: one case where $\mathcal{Z}$ and $\perp^{\bullet}[\mathcal{Z}]$ describe the same variable, and the other case where they do not.

Towards a Full Analysis

The main task when extending a sharing analysis based on a domain such as *Share* to a freeness analysis using $Share \times Free$ is to provide a suitable abstract unification algorithm. The introduction of a correct abstract unification algorithm for freeness analysis is an important motivation for our work. Our unification algorithm can be used together with techniques for combining abstract domains [6] to provide a full freeness analysis. In such an approach an analysis based on the domain *Share* is augmented with freeness information by converting the resulting pair to a set of abstract equation systems as described above. Our abstract unification algorithm is then applied to derive freeness information which is used to augment the result of the abstract unification using *Share*. Hence we obtain an element of $Share \times Free$. Other operations can be based on the more simple specification directly in $Share \times Free$. A similar approach can be taken to augment other domains with a freeness component.

Developing abstract equation systems into a full-fledged domain for a framework as described in [1] is a more involving task. The nontrivial burden is to define an order relation on sets of abstract equations systems satisfying the requirement of the framework: Let $S\mathcal{E}_1$ and $S\mathcal{E}_2$ be sets of abstract equation systems. Then $S\mathcal{E}_1 \leq S\mathcal{E}_2$ implies that the set of equation systems described by $S\mathcal{E}_1$ is included in the set of equation systems described by $S\mathcal{E}_2$. This paper has provided a starting point for the development of a complete freeness analysis, where the abstraction consists of a single abstract equation system. The least upper bound operation is a generalisation of anti-unification, and not simply set union as is the case for the domain of this paper. Details on the operations as well as an experimental evaluation can be found in [3, 18].

Conclusions

The paper presents a concise and correct abstract unification algorithm, providing the basis for a freeness analysis for logic programs. Our approach consists in carefully mimicking each step in a standard concrete unification algorithm. This allows us to obtain in a straightforward fashion a clear and intuitive algorithm together with a proof of its correctness. To the best of

our knowledge, this paper presents the first proof of correctness of a freeness analysis (which considers sharing information) for logic programs. Our approach is facilitated by a novel form of abstract domain termed abstract equation systems which consists of equations involving both concrete and abstract terms. This representation enables us to apply a methodology similar to that described in [4].

There are cases were the proposed abstract unification algorithm derives freeness information with a higher degree of precision than previous proposals. However, our abstract domain does lack several types of information such as *groundness* and *linearity*[3] which have a strong influence also on freeness information. Introducing an additional annotation $\perp^g$ on meta-terms to indicate a ground term is straightforward. Linearity information is harder to handle. It is the nondeterministic nature of our algorithm which is perhaps the main obstacle to a practical freeness analysis for logic programs. However, it is exactly this approach which enables us to derive a relatively simple, yet sufficiently precise abstract unification algorithm together with its proof of correctness. This is the contribution of the paper.

Ongoing work addresses the deficiencies mentioned above. A preliminary description of this work can be found in [2, 3] where abstract equation systems are enhanced to capture both groundness and linearity information. A full analysis and its experimental evaluation is reported in [18]. A deterministic abstract unification algorithm is obtained by choosing a specific concrete derivation to mimic depending on the structure of the abstract system. Confluence in the concrete algorithm justifies this approach. However, it is worth noting that both the algorithms as well as the proof sketches given in [2, 3, 18] are far more complicated than those presented here.

The abstract unification algorithm described in this paper is designed by mimicking the concrete algorithm of [17] which applies an occur check. An interesting direction for future research is to design an algorithm which instead considers rational trees such as the algorithm proposed by Colmerauer in [8].

Acknowledgement

This work was funded in part by the ESPRIT project 5246 PRINCE and by "Prog. Fin. Sistemi informatici e Calcolo Parallelo" of CNR, grant n.91.00026.69. The work was initiated while Michael Codish and Dennis Dams were visiting the University of Padova. During the course of the work Michael Codish was supported by a fellowship from KU Leuven. Maurice Bruynooghe is supported by the Belgian National Fund for Scientific Research. We thank A. Cortesi, M. García de la Banda, Manuel Hermenegildo and Renga Sundararajan for useful discussions. The constructive criticism of the anonymous referees is appreciated.

[3] A term is linear if every variable occurs at most once in it.

Appendix

Different occurrences of a meta-term $\bot[V]$ will sometimes be denoted $\bot_1[V]$, $\bot_2[V]$, etc.

The condition that a concrete or abstract rule modifies the system is only needed to prove termination. In the proofs below that do not concern termination (i.e., all except the proof of Theorem 5.1), it is convenient to drop it, that means, whatever abstract/concrete equation being selected in a system, an "invariant step" may always be applied. By this assumption, we avoid the need to consider separately the case that the abstract/concrete system remains invariant.

Proof of Lemma 5.1

Suppose that $E \rightarrow E' \neq fail$ and $\mathcal{E} \propto E$. We construct $\mathcal{E}'$ such that $\mathcal{E} \rightarrow \mathcal{E}'$ and $\mathcal{E}' \propto E'$. $\mathcal{E} \propto E$ implies that there exists an abstract term replacement μ, coherent with $\mathcal{E}$, such that $\mu(\mathcal{E}) = E$; fix this μ. Let e be the equation in E that is reduced, i.e., $E \equiv e :: \bar{E}$. Then there is $\varepsilon \in \mathcal{E}$ such that $\mu(\varepsilon) = e$, i.e., $\mathcal{E} \equiv \varepsilon :: \bar{\mathcal{E}}$. $\mathcal{E}'$ is constructed by indicating which rule of the abstract unification algorithm is applied to reduce ε. $\mathcal{E}' \propto E'$ is then shown by giving a μ', coherent with $\mathcal{E}'$, such that $\mu'(\mathcal{E}') = E'$. The proof is divided according to the structure of the abstract unification algorithm in Figure 4.1. P denotes some variable in $\mathcal{P}Var$; t, t_i and s_i are terms in $\mathcal{P}Term$, $\tau \in Term$ (so $\tau \notin \mathcal{M}Term$).

(1,2,3) $\varepsilon \equiv X = X$, $\varepsilon \equiv f(\tau_1, \ldots, \tau_n) = X$, or $\varepsilon \equiv f(\tau_1, \ldots, \tau_n) = f(\xi_1, \ldots, \xi_n)$. In these cases, e is respectively of the form $P = P$, $f(t_1, \ldots, t_n) = P$ or $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$. So, e is reduced by respectively a *remove*, *switch* or *peel* step. $\mathcal{E}'$ is constructed by applying the corresponding steps 1, 2 or 3 of the abstract unification algorithm to equation ε. μ' is taken equal to μ; clearly, μ' is coherent with $\mathcal{E}'$ and $\mu(\mathcal{E}') = E'$.

(4) $\varepsilon \equiv X = \tau$. This case follows in a similar way as case 5(d) below.

(5) $\varepsilon \equiv \tau = \bot[V]$. In this case, e may have one of the following forms.

(a,b) $e \equiv P = P$ or $e \equiv f(t_1, \ldots, t_n) = P$. So, e is reduced by respectively a *remove* or *switch* step. $\mathcal{E}'$ is constructed by applying the corresponding step 5(a) or 5(b) from the abstract unification algorithm to equation ε. μ' is taken equal to μ; clearly, μ' is coherent with $\mathcal{E}'$ and $\mu(\mathcal{E}') = E'$.

(c) $e \equiv f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$ and e is reduced by applying a *peel* step, yielding $E' \equiv \{t_1 = s_1, \ldots, t_n = s_n\} \cup \bar{E}$. Then $\tau = f(\tau_1, \ldots, \tau_n)$. Applying rule 5(c) of the abstract algorithm gives $\mathcal{E}' \equiv \{\tau_1 = \bot_1[V], \ldots, \tau_n = \bot_n[V]\} \cup \bar{\mathcal{E}}$. Define μ' to be the same as μ except that the new occurrences of $\bot[V]$ are mapped by $\mu'(\bot_i[V]) = s_i$ $(i = 1..n)$. It is straightforward to show that μ' is an abstract term replacement and that $\mu'(\mathcal{E}') = E'$. Moreover, let ξ and ξ' be occurrences of equated terms in $\mathcal{E}'$ such that $\mu'(\xi) = s$, $\mu'(\xi') = s'$ and $vars(s) \cap vars(s') \neq \emptyset$. Then: (1) if s and s' are left or right sides of equations in E then also $vars(\mu(\xi)) \cap vars(\mu(\xi')) \neq \emptyset$ and hence $vars(\xi) \cap vars(\xi') \neq \emptyset$; (2) if $s, s' \in \{s_1, \ldots, s_n\}$ then $vars(\xi) = vars(\xi') = V$ and as we assume that $V \neq \emptyset$, $vars(\xi) \cap vars(\xi') \neq \emptyset$; (3) if $s \in \{s_1, \ldots, s_n\}$ and s' is an occurrence of an equated term in E then $vars(s) \cap vars(s') \neq \emptyset \Rightarrow vars(f(s_1, \ldots, s_n)) \cap vars(s') \neq \emptyset$ which implies that $vars(\bot[V]) \cap vars(\xi') \neq \emptyset$. Hence $\mathcal{E}' \propto E'$.

(d) $e \equiv\ P = t$ and e is reduced by a *substitute* step, yielding $E' \equiv\ P = t\ ::\ \bar{E}[P/t]$. Then $\tau = X$, $\mu(X) = P$ and $\mu(\bot[V]) = t$. Applying rule 5(d) of the abstract algorithm gives $\mathcal{E}' \equiv\ X = \bot[V']\ ::\ \bar{\mathcal{E}}[X + \mathcal{Q}][X/\bot[V']]$, where $V' = V[X/\mathcal{Q}]$. Define μ' as follows: $\mu'(X) = P$, $\mu'(\bot[V']) = t$, and for every $\hat{\tau}$ in $\mathcal{E}$, $\mu'(\hat{\tau}[X + \mathcal{Q}][X/\bot[V']]) = \mu(\hat{\tau})[P/t]$. Then $\mathcal{E}' \propto E'$ under μ', as we show now.

By construction it is obvious that μ' is an abstract term replacement and that $\mu'(\mathcal{E}') = E'$. We show that μ' is coherent: let s_1 and s_2 be occurrences of terms in $P = t\ ::\ \bar{E}$ and consider the corresponding terms s_1' and s_2' after performing the substitution. Let ξ_1, ξ_2, ξ_1' and ξ_2' be such that $\mu(\xi_1) = s_1$, $\mu(\xi_2) = s_2$, $\mu(\xi_1') = s_1'$ and $\mu(\xi_2') = s_2'$. We have to prove that if s_1' and s_2' share, then ξ_1' and ξ_2' do. Clearly, if one of s_1' and s_2' is the P, it will not share with the other, as all occurences of P in $\bar{E}$ are eliminated by the substitute step. Now suppose s_1' and s_2' are occurrences in E' that share. Then $s_1' = s_1[P/t]$ and $s_2' = s_2[P/t]$. We distinguish the following cases: (1) s_1 and s_2 share. Because μ is coherent, ξ_1 and ξ_2 share, and because syntactic substitution preserves sharing between terms, also ξ_1' and ξ_2' share. (2) s_1 and s_2 do not share. Then it must be the case that P appears in (or is equal to) only one of them, say in s_1 (the other case is symmetric) and t and s_2 share. Because μ is coherent, $\bot[V]$ and ξ_2 share; furthermore, $X \in vars(\xi_1)$. So ξ_1' and ξ_2' share by construction.

(6) $\varepsilon \equiv\ \bot[V] = \tau$.

(a,b,c) The cases $e \equiv\ P = P$, $e \equiv\ f(t_1, \ldots, t_n) = P$ and $e \equiv\ f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$ are similar to cases 5(a,b,c).

(d) $e \equiv\ P = t$ and the rule applied is a *substitute*, yielding $E' \equiv\ P = t\ ::\ \bar{E}[P/t]$. Then $\mu(\bot[V]) = P$ and $\mu(\tau) = t$. We distinguish two cases:

(i) If $P \in V$, then rule 6(d)(i) should be applied with P for X, giving $\mathcal{E}' \equiv\ P = \tau\ ::\ \bar{\mathcal{E}}[P + \mathcal{Q}][P/\tau]$ (note that $P \in V$ and $P \in vars(\tau)$ implies that $P \in vars(t)$, in which case the occur check would lead to failure of the concrete unification; hence, we may assume $P \notin vars(\tau)$). Define μ' as follows: $\mu'(P) = P$, $\mu'(\tau) = t$ and for $\bar{\tau}$ in $\bar{\mathcal{E}}$, $\mu'(\bar{\tau}[P + \mathcal{Q}][P/\bar{\tau}]) = \mu(\bar{\tau})[P/t]$. It is easily verified that μ' is an abstract term replacement and that $\mu'(\mathcal{E}') = E'$. In a similar way as in case 5(d), it follows that μ' is coherent.

(ii) If $P \notin V$, then rule 6(d)(ii) should be applied. Let $s_1, \ldots, s_n$ be all occurrences of terms which contain P in $\bar{E}$, and let $\xi_1, \ldots, \xi_n$ be the corresponding terms in $\bar{\mathcal{E}}$, i.e., $\mu(\xi_i) = s_i$ for $i = 1..n$. Now let Y be a variable that is shared by all ξ_i, and choose this Y in rule 6(d)(ii), giving $\mathcal{E}' \equiv\ \mathcal{Q} = \tau\ ::\ \bar{\mathcal{E}}[Y + \mathcal{Q}][\mathcal{Q}/\tau]$. Define μ' as follows: $\mu'(\mathcal{Q}) = P$, $\mu'(\tau) = t$ and for $\bar{\tau}$ in $\bar{\mathcal{E}}$, $\mu'(\bar{\tau}[Y + \mathcal{Q}][\mathcal{Q}/\bar{\tau}]) = \mu(\bar{\tau})[P/t]$. It is easily verified that μ' is an abstract term replacement and that $\mu'(\mathcal{E}') = E'$. In a similar way as in case 5(d), it follows that μ' is coherent.

(7) $\varepsilon \equiv\ \bot[V_1] = \bot[V_2]$. Denote $V = V_1 \cup V_2$, $e \equiv\ s_1 = t_1$ and $\bar{E} = \{s_2 = t_2, \ldots, s_n = t_n\} \cup \hat{E}$ ($n \geq 1$). So there are occurrences $\bot_i[V_1]$ and $\bot_i[V_2]$ such that $\mu(\bot_i[V_1]) = s_i$ and $\mu(\bot_i[V_2]) = t_i$ ($i = 1..n$) and $\mu(\bar{\mathcal{E}}) = \hat{E}$. There are four cases depending on the structure of e:

(a) $e \equiv\ P = P$, which is reduced by a *remove*, giving $E' \equiv\ \{s_2 = t_2, \ldots, s_n = t_n\} \cup \hat{E}$. Construct μ' to map $2n-2$ occurrences of $\perp[V]$ to the terms s_i and t_i $(i = 2..n)$. Moreover, for every occurrence of an abstract term τ in $\bar{\mathcal{E}}$, μ' maps $\tau[\bigwedge_{P \in V} P/\perp[V]]$ to $\mu(\tau)$. It is straightforward to show that μ' is a coherent abstract term replacement and that $\mu'(\mathcal{E}') = E'$.

(b,c) $e \equiv\ f(t_1, \ldots, t_n) = P$ and $e \equiv\ f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$ are similar.

(d) $e \equiv\ P = t$. In this case μ' is constructed to map $2n$ occurrences of $\perp[V]$ respectively to the terms P, t, $s_i[P/t]$ and $t_i[P/t]$ $(i = 2..n)$. Moreover, for every occurrence of an abstract term τ in $\bar{\mathcal{E}}$, μ' maps $\tau[\bigwedge_{P \in V} P/\perp[V]]$ to $\mu(\tau)[P/t]$. It is easily verified that μ' is coherent and that $\mu'(\mathcal{E}') = E'$. □

Proof of Theorem 5.1

The proof is similar to that given in [17] for concrete unification where equation systems are associated with elements of the well-founded domain of triplets (n_1, n_2, n_3) with the lexicographical ordering, where n_1 is the number of "unsolved" variables (a variable is unsolved if it *does not* occur only once as the left-hand side of some equation), n_2 is the number of occurrences of function symbols, and n_3 the number of equations of the form $X = X$ or $t = X$ (where t is compound). In our case, a similar well-founded ordering is used on sextuples $(n_1, n_2, n_3, n_4, n_5, n_6)$, where:

- n_1 is the number of unsolved variables,
- n_2 is the number of unsolved variables which have occurrences outside meta-terms,
- n_3 is the number of occurrences of function symbols,
- n_4 is the number of abstract equations that *can describe* an equation of the form $X = X$ or $t = X$, and
- n_5 is the number of meta-terms,
- n_6 is the sum of the sizes of the meta-terms, where the size of a meta-term $\perp[V]$ in an equation system is the number of meta-variables occurring in the equation system but not in V.

Table .1 describes the effect that application of the rules of the abstract unification algorithm of Figure 4.1 has on the tuple $(n_1, ..., n_6)$. An entry "$<$" in this table means that the corresponding n_i is decreased, "$\leq$" means that the n_i either stays the same or is decreased. A "?" indicates that the effect on the corresponding n_i is not relevant. No entry means that the n_i stays the same.

For the rules 4, 5(d), 6(d)(i) and 7, a case distinction is made, as follows:

- 4: If X has no occurrences in meta-terms, then n_1 decreases; otherwise n_1 stays the same (because the new variable $\mathcal{Q}$ is introduced), but n_2 decreases because $\mathcal{Q}$ only occurs in meta-terms.

	n_1	n_2	n_3	n_4	n_5	n_6
1	$\leq$	$\leq$		$<$	?	?
2				$<$	?	?
3			$<$	?	?	?
4	$<$	?	?	?	?	?
4		$<$	?	?	?	?
5(a)	$\leq$	$\leq$		$<$	?	?
5(b)				$<$	?	?
5(c)	$\leq$	$\leq$	$<$	?	?	?
5(d)	$<$	?	?	?	?	?
5(d)		$<$	?	?	?	?
6(a)	$\leq$	$\leq$		$<$	?	?
6(b)	$\leq$	$\leq$		$<$	?	?
6(c)			$<$	?	?	?
6(d)(i)	$<$	?	?	?	?	?
6(d)(i)		$<$	?	?	?	?
6(d)(ii)					$<$	?
7		$<$	?	?	?	?
7						$<$

Table .1: Effect of the rules on the n_i.

- 5(d): If X has no occurrences in meta-terms *and is not in* V, then n_1 decreases; otherwise n_1 stays the same (because the new variable $\mathcal{Q}$ is introduced), but n_2 decreases because $\mathcal{Q}$ only occurs in meta-terms.
- 6(d)(i): If X has no other occurrences in meta-terms, then n_1 decreases; otherwise n_1 stays the same (because the new variable $\mathcal{Q}$ is introduced), but n_2 decreases because $\mathcal{Q}$ only occurs in meta-terms.
- 7: If there are variables from V that have occurrences *outside* meta-terms, then n_2 decreases (while n_1 stays the same); otherwise, as the rule does not leave the system invariant, n_6 decreases.

As for rule 6(d)(ii), n_1 and n_2 stay the same because although the new variable $\mathcal{Q}$ is introduced, it appears "solved".

From the table it is clear that the application of any rule decreases the sextuple $(n_1, n_2, n_3, n_4, n_5, n_6)$ in the lexicographical ordering.

It can easily be seen that this same variant function can be used to prove that Theorem 5.1 also holds for the more concise algorithm of Figure 4.2. □

Proof of Theorem 5.2

An abstract equation system in solved form can represent a concrete system which is not in solved form. This is possible in the presence of one or more equations of the form $\bot[V] = \bot[V]$

which possibly represent the unsolved part of a concrete system. This intuition leads to the following lemma:

Lemma .1 *If $\mathcal{E}$ is in solved form, $\mathcal{E} \propto E$ and $mgu(E) \neq fail$ then $\mathcal{E} \propto mgu(E)$.*

PROOF. By induction on the number k of steps needed to obtain $mgu(E)$ from E using the concrete unification algorithm of Figure 2.1 (under the assumption that each such step modifies the equation system).
base: $k = 0$ trivial as $E = mgu(E)$.
step: Assume $\underline{e} :: E \rightarrow E' \rightarrow^k mgu(E')$, i.e., equation e is chosen for reduction. Assume the abstract system is of the form $\varepsilon :: \mathcal{E}$ and $\mu(\varepsilon) = \{e, e_1, \ldots, e_n\}$, $n \geq 0$, $\mu(\mathcal{E}) = E \setminus \{e_1, \ldots, e_n\}$. We show the existence of some coherent μ' such that $\mu'(\varepsilon :: \mathcal{E}) = E'$. We distinguish the following cases according to the form of equation e and the corresponding step being taken in the concrete unification.

1. $e \equiv X = X$, the concrete step is *remove*. Since no rule can modify $\varepsilon :: \mathcal{E}$, ε must have the form $\bot[V] = \bot[V]$. So $\mu(\varepsilon) = \{X = X, e_1, \ldots, e_n\}$. Define μ' by $\mu'(\varepsilon) = \{e_1, \ldots, e_n\}$ and $\mu'(\mathcal{E}) = \mu(\mathcal{E})$. Then clearly $\mu'(\varepsilon :: \mathcal{E}) = E'$ and μ' is coherent.

2. $e \equiv f(t_1, \ldots, t_m) = X$, the concrete step is *switch*. Since no rule can modify $\varepsilon :: \mathcal{E}$, ε must have the form $\bot[V] = \bot[V]$. So $\mu(\varepsilon) = \{f(t_1, \ldots, t_m) = X, e_1, \ldots, e_n\}$. Define μ' by $\mu'(\varepsilon) = \{X = f(t_1, \ldots, t_m), e_1, \ldots, e_n\}$ and $\mu'(\mathcal{E}) = \mu(\mathcal{E})$. Then clearly $\mu'(\varepsilon :: \mathcal{E}) = E'$ and μ' is coherent.

3. $e \equiv f(t_1, \ldots, t_m) = f(s_1, \ldots, s_m)$, the concrete step is *peel*. Since no rule can modify $\varepsilon :: \mathcal{E}$, ε must have the form $\bot[V] = \bot[V]$. So $\mu(\varepsilon) = \{f(t_1, \ldots, t_m) = f(s_1, \ldots, s_m), e_1, \ldots, e_n\}$. Define μ' by $\mu'(\varepsilon) = \{t_1 = s_1, \ldots, t_m = s_m, e_1, \ldots, e_m\}$, and $\mu'(\mathcal{E}) = \mu(\mathcal{E})$. Then clearly $\mu'(\varepsilon :: \mathcal{E}) = E'$ and μ' is coherent.

4. $e \equiv X = t$, the concrete step is *subst*. The concrete step modifies the system, so $X \in vars(f)$ with f an equation in E. In the following, $\tau \in Term$.

   $e :: E$ is not described by $Y = \tau :: \mathcal{E}$ (with $\mu(Y) = X$) because then f is described by $\mathcal{E}$, so $Y \in vars(\mathcal{E})$ and rule 4 would modify $\mathcal{E}$ (Y is either X or a meta-variable). Also, $Y \in vars(\tau)$ implies that $\mu(Y)$ (which equals X) is in $vars(\mu(\tau))$. That is inconsistent with the assumption that the concrete system does not fail.

   Similarily, $e :: E$ is not described by $Y = \bot[V] :: \mathcal{E}$ with $\mu(Y) = X$ because then f is described by $\mathcal{E}$, so $Y \in vars(\mathcal{E})$ and rule 5(d) would modify $\mathcal{E}$.

   Similarily, $e :: E$ is not described by $\bot[Y, \ldots] = \tau :: \mathcal{E}$ with $\mu(Y) = \mu(\bot[Y, \ldots]) = X$ because then f is described by $\mathcal{E}$, so $Y \in vars(\mathcal{E})$ and rule 6(d)(i) would modify $\mathcal{E}$.

   (In the above 3 cases the rule would introduce a $\mathcal{Q}$.)

   So ε must have the form $\bot[V] = \bot[V]$ and $\mu(\varepsilon) = \{X = t, e_1, \ldots, e_n\}$. Moreover, the system cannot be modified by application of rule 7 on ε, so any toplevel term containing X in a concrete equation f in E must be represented by a meta-term $\bot[V']$ which remains invariant under application of rule 7 on ε, so $V' \supseteq V$. Define $\mu'(\varepsilon) = \{X = t, e_1[X/t], \ldots, e_n[X/t]\}$; for meta-variables $\mathcal{Z}$: $\mu'(\mathcal{Z}) = \mu(\mathcal{Z})[X/t] = \mu(\mathcal{Z})$; for meta-term occurrences $\bot[V']$: $\mu'(\bot[V']) = \mu(\bot[V'])[X/t]$. Then clearly $\mu'(\varepsilon :: \mathcal{E}) = E'$.

Finally, we show that μ' is coherent. Assume s'_1 and s'_2 are left- or right-hand-sides in E' described by τ_1 and τ_2 in $\mathcal{E}$. Let s_1 and s_2 be the corresponding elements in E, i.e., $s'_i = s_i[X/t]$. Notice that s_1 and s_2 are also described by τ_1 and τ_2 respectively, as $\mathcal{E}$ describes both E and E'. Coherence of μ' requires that τ_1 and τ_2 share when s'_1 and s'_2 do. Assume s'_1 and s'_2 share. Either s_1 and s_2 already shared, in which case τ_1 and τ_2 also share by coherence of μ. Or they did not in which case one of them (say s_1) is X or contains X, so $\tau_1 = \bot[V']$ with $V \subseteq V'$ and the other (s_2) shares with t; as t is described by $\bot[V]$, it means τ_2 shares with $\bot[V]$, so it also shares with $\bot[V'] = \tau_1$. □

The proof of Theorem 5.2 now proceeds as follows. Suppose that $\mathcal{E} \propto E$ and $mgu(E) \neq fail$. By the latter, there exists a step $E \rightarrow E'$ (possibly leaving the system invariant). If furthermore the possible steps of $\mathcal{E}$ are $\mathcal{E} \rightarrow \mathcal{E}_j (j = 1, \ldots, l)$, then by Lemma 5.1 there exists a j such that $\mathcal{E}_j \propto E'$.

By simple induction it can now be shown that if $\mathcal{E}$ has solved forms $\mathcal{E}_1, \ldots, \mathcal{E}_n$ then for some $i \in \{1, .., n\}$, $\mathcal{E}_i$ is reached in m steps and $E \rightarrow^m E''$ such that $\mathcal{E}_i \propto E''$. By Lemma .1 it then follows that $\mathcal{E}_i \propto mgu(E'') = mgu(E)$. □

Proof of Theorem 5.3

Theorem 5.3 follows directly from Theorem 5.2 and the following

Lemma .2 *If $\mathcal{E} \propto E$ and $\mathcal{E}$ is in solved form then freeness of X in $\mathcal{E}$ implies freeness of X in E.*

PROOF. $\mathcal{E}$ is in solved form, so by Lemma .1, $\mathcal{E}$ describes $mgu(E)$. Suppose X is free in $\mathcal{E}$ but not free in $mgu(E)$, then $mgu(E)$ contains an equation $X = f(t_1, \ldots, t_n)$. This equation is described via some μ by an equation $\tau_1 = \tau_2$ in $\mathcal{E}$ ($\tau_1, \tau_2 \in \mathcal{A}Term$). So τ_2 must be of the form $f(\tau_{2,1}, \ldots, \tau_{2,n})$ or of the form $\bot[V]$ with $\mu(\bot[V]) = f(t_1, \ldots, t_n)$. τ_1 is either X (it cannot be a meta-variable as the fact that X is free in $\mathcal{E}$ implies $X \in vars(\mathcal{E})$) or of the form $\bot[V]$ with $X \in V$. All these possibilities contradict the assumption that X is free in the solved form $\mathcal{E}$. □

References

[1] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10(2):91–124, Feb. 1991.

[2] 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*, number 724 in Lecture Notes in Computer Science, pages 153–164. Springer-Verlag, 1993.

[3] M. Bruynooghe, M. Codish, and A. Mulkers. Abstract unification for a composite domain deriving sharing and freeness properties of program variables. Workshop on Verification and Analysis of (Concurrent) Logic Languages, ICLP94, June 1994.

[4] M. Codish, D. Dams, and E. Yardeni. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In Furukawa [11], pages 79–93.

[5] M. Codish, D. Dams, G. Filé, and M. Bruynooghe. Freeness analysis for logic programs — And correctness? In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 116–131, Budapest, Hungary, 1993. The MIT Press.

[6] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 194–205. ACM Press, 1993. To appear in *ACM Trans. Prog. Lang. Syst.*

[7] A. Cortesi and G. Filé. Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-based Program Transformation*. ACM Press, 1991.

[8] A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S. A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, 1982.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Jan. 1977.

[10] S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Trans. Prog. Lang. Syst.*, 11(3):418–450, July 1989.

[11] K. Furukawa, editor. *Proceedings of the Eighth International Conference on Logic Programming*, Paris, France, 1991. The MIT Press.

[12] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of global analysis in strict independence-based automatic program parallelization. In *International Symposium on Logic Programming*, pages 320–336, Nov. 1994. MIT Press.

[13] M. V. Hermenegildo and K. J. Greene. &-Prolog and its performance: Exploiting independent And-Parallelism. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268, Jerusalem, 1990. The MIT Press.

[14] D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *The Journal of Logic Programming*, 13(2 and 3):291–314, 1992.

[15] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, CA, 1988.

[16] K. Marriott and H. Søndergaard. Analysis of constraint logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 531–547, Austin, 1990. ALP, MIT Press.

[17] A. Martelli and U. Montari. An efficient unification algorithm. *ACM Trans. Prog. Lang. Syst.*, 4(2):258–282, Apr. 1982.

[18] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the practicality of abstract equation systems. In L. Sterling, editor, *Proceedings of the International Conference on Logic Programming*, pages 781–795, Kanagawa, Japan, 1995. MIT Press.

[19] K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In Furukawa [11], pages 49–63.

[20] H. Søndergaard. An application of abstract interpretation of logic programs: Occur-check reduction. In *Proceedings of European Symposium on Programming '86*, number 213 in *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, 1986. (extended abstract).

[21] R. Sundararajan and J. Conery. An abstract interpretation scheme for groundness, freeness and sharing analysis of logic programs. In *Conference on Foundations of Software Technology and Theoretical Computer Science*, number 652 in Lecture Notes in Computer Science, pages 203–216. New Delhi, Springer-Verlag, Dec. 1992.