Abstracting Unification: A Key Step in the Design of Logic Program Analyses

Maurice Bruynooghe¹ and Michael Codish² and Anne Mulkers¹

¹ K.U.Leuven, Department of Computer Science,
Celestijnenlaan 200A, 3001 Heverlee, Belgium

² Ben-Gurion University, Department of Mathematics and Computer Science,
P.O.B. 653, 84105 Beer-Sheva, Israel

Abstract. This paper focuses on one of the key steps in the design of semantic based analyses for logic programs — the definition of an abstract unification algorithm for a given notion of data description. We survey some of the major notions of data descriptions proposed in the context of mode and sharing analyses. We demonstrate how a careful and systematic analysis of the underlying concrete unification algorithm contributes to the design of the abstract algorithm. Several relevant properties of concrete substitutions which influence the design of abstract domains and algorithms are described. We make use of a novel representation called abstract equation systems to uniformly represent a wide range of data descriptions for such analyses proposed in the literature.

1 Introduction

Abstract interpretation has been established by Cousot and Cousot [17] as a formal basis for program analysis. The essence is captured by the statement: “Abstract interpretation is computation over a non-standard domain of data descriptions”. The computation performed during the execution of a logic program consists basically of a sequence of unifications. Similarly, an abstract interpretation-based program analysis consists of a sequence of unifications over a non-standard domain of data descriptions. There is more to it however, as concrete computations potentially perform an unbounded number of unifications, thereby accumulating substitutions over an unbounded number of variables. The problems arising from the unboundedness are solved by abstract interpretation frameworks [5, 6, 10, 18, 24, 31, 33] which are designed in such a way that a safe approximation of the program behavior — modeled as a fixpoint of a function over the standard domain of the program — can be obtained in a finite number of iterations. When applying such a framework to obtain program analyses, an appropriate domain of data descriptions and a number of primitive operations must be provided. The key component among these operations is unification in the domain of data descriptions, called abstract unification for short. Many of the more significant applications of such frameworks to program analysis focus on characterizing properties of the terms which program variables may be bound to during executions of the program. Properties of interest include groundness, freeness, linearity, and sharing — the precise definitions of which will be clarified

below. These properties are useful for a wide range of compiler optimizations, for parallelizing the execution of logic programs as well as for improving the precision of many other types of program analyses. As such, the relevant literature is saturated with proposals for various domains and corresponding unification algorithms which provide the basis for program analyses to obtain information regarding these types of properties. Developing a correct abstraction of unification is for many abstract domains a difficult, if not to say error-prone process [8].

In this paper we discuss some of the major domains of data descriptions and their corresponding abstract unification algorithms. We focus mainly on the issue of ensuring correctness of abstract unification. We develop a setting for analyzing the correctness of abstract unification which is general enough to cover a wide range of analyses described in the literature. We argue that the essential step in developing a correct abstraction is the careful analysis of the underlying concrete unification, we describe several relevant properties of concrete substitutions and their relationship, and finally we unravel the intuitions underlying the abstract unification operation of several abstract domains.

Having argued that computation consists of a sequence of unifications and having pointed out that the problems posed by the unboundedness of the sequences are handled by frameworks, we focus the rest of the paper on abstracting a single unification, which is the key step in the design of a logic program analysis. Section 2 recalls basic concepts about substitutions. Section 3 contains a formulation of abstract unification which is suited as a basis for abstract interpretation frameworks. Section 4 reviews a set of properties of interest for program analyses. In Section 5 the importance of sharing information for both correctness and precision of program analyses is discussed. Section 6 provides a more general formulation of abstract unification that is based on the notion of equation systems. In Section 7, a methodology for developing correct analyses is discussed and the ideas underlying soundness proofs of a few well-known abstract domains are reviewed. In Section 8, a domain of abstract equation systems is described as an illustration of an abstract unification algorithm designed according to this methodology.

2 Preliminaries

We assume the standard notation for logic programs as described for example in [2, 32]. Let Σ be a fixed alphabet of function symbols and Var a denumerable set of concrete variables. The set of terms constructed using function symbols from Σ and variables from Var is denoted Term . The set of variables occurring in a syntactic object s is denoted by $\text{Var}(s)$.

Substitutions are mappings from Var to Term which behave as the identity mapping almost everywhere and which are described by a set of pairs X/t . The domain of a substitution θ is denoted $\text{Dom}(\theta) = \{X \mid X\theta \neq X\}$ and $\text{Range}(\theta) = \{X \in \text{Var}(Y\theta) \mid Y \in \text{Dom}(\theta)\}$ is the range of θ . Unification algorithms produce idempotent substitutions, so we restrict our attention to such substitutions. The composition of two substitutions θ and σ is denoted by $\theta\sigma$ and its effect on an

expression is the same as that of first applying θ and then σ . The restriction of a substitution to a set of variables S is $\theta|S = \{X/t \in \theta \mid X \in S\}$. A unifier θ of two terms t and s is a substitution such that $t\theta = s\theta$ and is called relevant if $\text{Var}(\theta) \subseteq \text{Var}(t = s)$ [2].

Program analyses typically focus on properties of a finite set of variables under a (set of) substitution(s). For example, on properties of a clause's variables under the set of substitutions corresponding to the states in which that clause can participate in a resolution step; or the properties of the variables in an atom under the set of its answer substitutions. This finite set of variables of interest is called the *reference domain* and its variables the *reference variables*. Throughout the paper V denotes a reference domain. A preorder with respect to a given reference domain can be defined over substitutions as follows: $\theta \leq_V \sigma$ iff there exists a substitution μ such that $\theta|V = (\sigma\mu)|V$. The preorder on substitutions induces a corresponding equivalence relation on substitutions which is denoted by $\equiv_V$. Note that for every substitution θ , $\theta \equiv_V \theta|V$ and there always exists a substitution σ with $\text{Dom}(\sigma) = V$ such that $\sigma \equiv_V \theta$. For example the empty substitution, ϵ , satisfies $\epsilon \equiv_V \{X_1/U_1, \dots, X_n/U_n\}$ where $V = \{X_1, \dots, X_n\}$ is disjoint from $\{U_1, \dots, U_n\}$. For a variable $X \in V$, the properties we focus on in analyses such as freeness, groundness, and sharing with other reference variables are invariant modulo $\equiv_V$. This means that $X\sigma$ and $X\theta$ have identical properties of interest when $\sigma \equiv_V \theta$. Hence, for program analyses we restrict our attention to substitutions modulo $\equiv_V$.

The Martelli-Montanari unification algorithm [35] can be applied to produce idempotent and relevant most general unifiers. We fix a partial function which maps a finite set of equations E to a solved form $mgu(E)$. An idempotent and relevant substitution, which is a most general unifier of E , corresponds to this solved form [2, 28]. A reference to $mgu(E)$ implicitly implies that E is satisfiable. In the following we abuse notation denoting by $mgu(E)$ the solved form of E as well as the corresponding most general unifier. For a substitution θ , we let $eqn(\theta)$ denote the corresponding set of equations (in solved form). For a finite set of equations E and an equation $e \notin E$, we let $e :: E$ denote $\{e\} \cup E$.

3 A General Formulation of Abstract Unification

The concrete operation which is to be mimicked over the abstract domain involves an equation (or set of equations) $s = t$ and a substitution σ . The unification to be performed is between $s\sigma$ and $t\sigma$; the result of the concrete operation is $\mu = \sigma mgu(\{s = t\}\sigma)$. This operation is termed a *unification step* in our setting and its abstract counterpart, *abstract unification*. In the context of a computation, s and t originate from atoms in a program, σ corresponds to a computation state specifying an instance of s and t , while μ is the next computation state. In the context of this paper we will assume that the reference domain, V , includes the variables of s and t . If needed, V can be extended to establish this. Abstractions of operations modifying the reference domain are not considered here. Moreover, as we consider substitutions modulo $\equiv_V$, we will assume that

$Dom(\sigma) \subseteq V$ and that μ is defined as $(\sigma mgu(\{s = t\}\sigma))|V$.

Abstract unification operates on data descriptions. A data description d_{in} describes for a particular analysis relevant properties of the reference variables under a computation state σ . Given a description d_{in} and an equation (or set of equations) $s = t$, abstract unification should provide a description d_{out} expressing a safe approximation of the relevant properties of the reference variables under the new computation state $\mu = (\sigma mgu(\{s = t\}\sigma))|V$ for all σ described by d_{in} .

In the standard framework of abstract interpretation [17, 18] the relation between data and data description is formalized in terms of a *Galois connection*. A Galois connection is a quadruple (C, α, D, γ) where:

1. $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ are complete lattices called *concrete* and *abstract domains* respectively;
2. $\alpha : C \rightarrow D$ and $\gamma : D \rightarrow C$ are monotonic functions called *abstraction* and *concretization functions* respectively; and
3. $\alpha(\gamma(d)) \sqsubseteq_D d$ and $c \sqsubseteq_C \gamma(\alpha(c))$ for every $d \in D$ and $c \in C$.

In practice it is often sufficient to specify only γ (or α). Other frameworks exist that impose less stringent conditions on the domains C , D , and the mappings defining their relationship (see [33] for a discussion). In the setting of the current paper: C is a set of concrete equation systems ordered by set inclusion; γ gives meaning to the data descriptions of D ; and γ together with the order of the concrete domain induce an order relation on the abstract domain. A description relation corresponding to the concretization mapping is defined by $d \propto c$ iff $c \in \gamma(d)$.

4 Properties of Interest for Program Analyses

The properties of interest for program analyses characterize the terms a reference variable may be bound to under a (set of) substitution(s). To formalize them, we introduce the concept of variable multiplicity which specifies the number of occurrences of a variable U in a term t under a substitution θ :

- $multiplicity_\theta(U, t) =$ the number of occurrences of U in $t\theta$.

This concept is not useful by itself as it makes reference to variables in the range of θ and as such is not invariant under $\equiv_V$. However, it is the basis for the notion of variable multiplicity for a term t under a set of substitutions Θ :

- $multiplicity_\Theta(t) = \max(\{multiplicity_\theta(U, t) \mid \theta \in \Theta, U \in \text{Var}(t\theta)\})$.

Based on the concept of variable multiplicity, two derived concepts are defined:

- $ground_\Theta(t) \Leftrightarrow multiplicity_\Theta(t) = 0$;
- $linear_\Theta(t) \Leftrightarrow multiplicity_\Theta(t) \leq 1$.

Groundness is a very important property and is expressed in almost every abstract domain. Linearity is not so important by itself, but as will be illustrated below, it can be used to increase the precision of abstract unification for other types of information and it is an important component in techniques for occur-check reduction [3, 41, 42].

A second basic predicate concerns the freeness of a term under a substitution θ or set of substitutions Θ :

- $free_\theta(t) \Leftrightarrow t\theta \in \text{Var}$;
- $free_\Theta(t) \Leftrightarrow \forall \theta \in \Theta. free_\theta(t)$.

Freeness captures a basic mode property of terms but is notoriously difficult to integrate correctly in an abstract domain [8, 40]. Note that $free_\Theta(t) \rightarrow multiplicity_\Theta(t) = 1$, and hence $free_\Theta(t) \rightarrow linear_\Theta(t)$.

Another important property for program analyses is the notion of sharing between terms under a substitution or set of substitutions as captured by:

- $share_\theta(t_1, t_2) \Leftrightarrow \text{Var}(t_1\theta) \cap \text{Var}(t_2\theta) \neq \emptyset$;
- $share_\Theta(t_1, t_2) \Leftrightarrow \exists \theta \in \Theta. share_\theta(t_1, t_2)$.

An important related concept characterizes the independence of terms:

- $indep_\Theta(t_1, t_2) \Leftrightarrow \neg share_\Theta(t_1, t_2)$.

Independence is a very important property in work on AND-parallelization of logic programs [21, 23].

The above notion of sharing is often referred to as *pair sharing* since it specifies the sharing of a pair of terms. Together with groundness and linearity it forms the pair-sharing domain of [9, 42]. A pair $(X, Y) \in V^2$ represents that X and Y may be bound to terms with common variables. A reflexive pair (X, X) represents possibly non-linear bindings. However in [16] reflexive pairs are used to represent possible non-ground bindings. Another useful notion is that of *set sharing* which is specified in terms of sharing groups, which group the variables of the reference domain V sharing a common variable under a substitution. The basic concept, which is not invariant under $\equiv_V$, is

- $sharegroup(\theta, U) = \{X \in V \mid U \in \text{Var}(X\theta)\}$.

For the purpose of program analysis, sharing groups with respect to arbitrary variables of sets of substitutions are collected, leading to the following derived concept:

- $sharegroups(\Theta) = \left\{ S \mid \begin{array}{l} \exists \theta \in \Theta, \exists U \in \text{Var}. \\ S = sharegroup(\theta, U) \neq \emptyset \end{array} \right\}$.

Sharegroups were first used by Jacobs and Langen [22, 23] and are known as the set-sharing domain. They were also studied and implemented by Muthukumar and Hermenegildo who also extended them with a freeness component [39, 40]. An element of the abstract domain is a set of sets of variables from V [22, 23,

26, 27, 39, 40]. Each set of reference variables S in a description SS is a sharing group expressing that the variables in S can share one or more variables in a substitution described by SS . A singleton set $\{X\} \in SS$ represents that the binding for X may contain a variable that does not occur in bindings to other reference variables.

Interesting properties relating sharing groups with some of the notions introduced above are:

- $\forall X \in V.$
 $ground_{\Theta}(X) \Leftrightarrow \forall S \in sharegroups(\Theta). X \notin S;$
- $\forall X, Y \in V.$
 $indep_{\Theta}(X, Y) \Leftrightarrow \forall S \in sharegroups(\Theta). X \in S \rightarrow Y \notin S;$
- $\forall t_1, t_2$ such that $Var(\langle t_1, t_2 \rangle) \subseteq V.$
 $indep_{\Theta}(t_1, t_2) \Leftrightarrow \forall X \in Var(t_1) \forall Y \in Var(t_2). indep_{\Theta}(X, Y).$

Finally, a basic predicate for expressing so-called covering information is introduced. A syntactic object O_1 covers another syntactic object O_2 under a substitution θ if $ground_{\theta}(O_1) \Rightarrow ground_{\theta}(O_2)$.

- $covers_{\theta}(O_1, O_2) \Leftrightarrow Var(O_1\theta) \supseteq Var(O_2\theta);$
- $covers_{\Theta}(O_1, O_2) \Leftrightarrow \forall \theta \in \Theta. covers_{\theta}(O_1, O_2).$

The notion of covering provides an alternative definition for the groundness predicate:

- $ground_{\Theta}(t) \Leftrightarrow covers_{\Theta}(\emptyset, t).$

The notion of covering is easily generalized to express implications between other types of properties. For example, in [12], various covering properties are applied to determine nonsuspension for concurrent logic programs. The notion of covering is the basis of the Prop domain [4, 11, 30, 34, 43], where a basic relation $covers_{\theta}(\langle X_1, \dots, X_n \rangle, Y)$ is expressed as a propositional formula $X_1 \wedge \dots \wedge X_n \rightarrow Y$.

Sharing groups also encode covering information. As an illustration consider the following example: let

$$sharegroups(\Theta) = \{\{X, Z\}, \{Y, Z\}, \{Y, Z, W\}, \{X\}\}.$$

It follows from the above definitions that for all $\theta \in \Theta$, any variable in $Z\theta$ must occur in either $X\theta$ or in $Y\theta$ and hence $covers_{\theta}(\langle X, Y \rangle, Z)$. Similarly, any variable in $W\theta$ or in $Y\theta$ is also in $Z\theta$, hence $covers_{\theta}(Z, \langle W, Y \rangle)$. A property useful to extract covering information from a set of sharing groups is the following: Let O be a syntactic object such that $Var(O) \subseteq V$ and $X \in V$, then

- $covers_{\Theta}(O, X) \Leftrightarrow \forall S \in sharegroups(\Theta). X \in S \rightarrow Var(O) \cap S \neq \emptyset.$

While the above relationship shows that sharing groups capture the covering information expressed in the domain Prop, it does not imply that a set-sharing analysis derives the same covering information as a Prop analysis. Indeed, it

is known that the abstract unification for the set-sharing domain may lose covering information which is preserved by the corresponding operation on the Prop domain [15].

Both pair- and set- sharing are related to independence information. However, the corresponding abstract domains are not directly comparable. On the one hand, the pair-sharing domain expresses also linearity information which enables the corresponding abstract unification algorithm to provide more precise independence information. On the other hand, the set-sharing domain captures covering information which helps propagate more precise groundness information. Combining both approaches [13] is often beneficial.

5 On the Importance of Sharing Information

A reader unfamiliar with the area might wonder why the tricky concept of sharing gets so much attention. It is not only because the derived concept of independence plays a crucial role in work on parallelization of logic programs, but because of a fundamental problem regarding correctness and precision of a domain having a mode component but no sharing component. In the following example, we consider such a simple domain which associates one of the modes *ground*, *free*, or *any* with each variable of the reference domain V (mode *any* for X in a data description d does not impose any constraint on $X\theta$ where $\theta \in \gamma(d)$).

Example 1. Let $V = \{X, Y\}$ and consider the unification $X = a$ together with a data description d_{in} which specifies that X and Y are *free*. A substitution σ described by d_{in} is $\{X/U, Y/W\}$. Performing the concrete unification under σ yields $\mu = \{X/a, Y/W\}$, i.e., X becomes *ground* while Y remains *free*. However also $\{X/U, Y/U\}$ is described by d_{in} and the corresponding concrete unification results in $\mu = \{X/a, Y/a\}$, i.e., both X and Y become *ground*. Any correct abstract unification algorithm must provide a data description d_{out} which assigns mode *any* to Y (and in general to any reference variable which is not *ground*). The mode domain of this example is so weak that all freeness information is lost under a single abstract unification involving a non-*free* term.

The problem is less acute if so-called substitution closed domains are used. In such domains, $\sigma \in \gamma(d) \Rightarrow \sigma\theta \in \gamma(d)$ for all θ . A domain assigning modes *ground* or *any* to the variables in the reference domain has this property. In this case, a correct result can be obtained when changing the modes only of those variables directly involved in the abstract unification. Let us reconsider Example 1 in the context of a description d_{in} which assigns mode *any* to both X and Y . Abstract unification for the equation $X = a$, may safely focus only on X providing a description d_{out} which assigns mode *ground* to X . The variable Y remains *any*, which is correct and, with the available information, is the best that is achievable since the abstract operation cannot know whether X and Y are bound to the same term. So, the problem of poor precision remains. Debray [19] was the first to recognize these problems and to point out a flaw in the ad-hoc

mode analysis of Mellish [36, 37] that used a domain which was not substitution closed.

Increased precision can be realized by introducing a technique termed re-execution [6, 29] into the framework. A more conventional approach is to enhance the abstract domain with a sharing component. A sufficient condition for not changing the mode of a variable X during an abstract unification is the following: given the description d_{in} and the abstract unification $s = t$, we must guarantee that for every $\sigma \in \gamma(d_{in})$, $\text{Var}(X\sigma) \cap \text{Var}(\{s = t\}\sigma) = \emptyset$, i.e., $\text{indep}_\sigma(X, \langle s, t \rangle)$. This implies that $X\sigma = X\mu$ for $\mu = \sigma mgu(\{s = t\}\sigma) \upharpoonright V$ and hence that the mode for X need not change in d_{out} . This motivates the need for sharing analyses.

As mentioned in Section 4, the *share*, *sharegroups*, and *covers* predicates have been used as a basis to express sharing in abstract domains. An extensive discussion of early variable-dependencies analyses can be found in [20].

6 A More General Formulation of Abstract Unification

This section describes a more general setting for abstract unification which covers both traditional analyses such as set-sharing and pair-sharing, as well as more recent ones using abstract equation systems [7, 8, 38]. This setting enables us to break up an abstract unification operation into smaller rewrite steps, much the same as the way the Martelli-Montanari unification algorithm breaks up a concrete unification. This approach facilitates both design and justification of abstract unification algorithms as correctness boils down to showing that the rewriting terminates in a “solved form” and that each rewrite step is correct.

In this approach, an abstract state is a pair $\langle \mathcal{E}, d \rangle$ consisting of a finite set of equations $\mathcal{E}$ and an annotation d typically providing mode and sharing information for some of the variables in $\mathcal{E}$. The description relation $\propto$ associates d with a substitution σ with solved form $eqn(\sigma)$ and $\mathcal{E}$ with a system of equations E . In terms of equations we will write this as $\langle \mathcal{E}, d \rangle \propto E \cup eqn(\sigma)$. In traditional analyses d describes properties of the variables in the reference domain V , $\text{Dom}(\sigma) \subseteq V$, $\mathcal{E}$ is a set of equations over the reference variables, and the relation between $\mathcal{E}$ and E is the identity relation. In the solved form of the abstract unification algorithm, the equation component is the empty set of equations, while the annotation d provides the updated properties of the reference variables.

In analyses based on abstract equation systems, a solved form has an equation component AE involving both reference variables as well as *abstract variables* and an annotation component d specifying properties of the abstract variables. So, a solved form is a pair $\langle AE, d \rangle$. The description relation associates such a pair with a set of equations $E \cup eqn(\sigma)$ where E is derived from AE by replacing the abstract variables by variables outside the reference domain and $\text{Dom}(\sigma)$ a subset of these newly introduced variables.

Example 2. Consider a set of equations $AE = \{X = f(@_1), Y = @_2\}$, where $V = \{X, Y\}$ and $@_1, @_2$ are abstract variables. The description $d = \{\{ @_1 \}, \{ @_1, @_2 \}\}$

is a set-sharing component. A concrete system described by AE obtained by mapping abstract variable $@_1$ to V_1 and $@_2$ to V_2 , is for instance $E = \{X = f(V_1), Y = V_2\}$. A substitution described by d is $\sigma = \{V_1/g(U_1, U_2), V_2/h(U_2)\}$. The equation system $E\sigma = \{X = f(g(U_1, U_2)), Y = h(U_2)\}$ gives the bindings of the reference variables. More details about the domain are in Section 8.

Using the abstract equations of [8], consider system $AE = \{X = \perp^+[U, W], Y = \perp[W]\}$. The d -component is implicit in AE and expresses that the first abstract variable $\perp^+[U, W]$ is bound to a non-variable term which can have variables not occurring elsewhere but also variables occurring in the binding of the second abstract variable $\perp[W]$, which can be bound to any term. So for example, mapping $\perp^+[U, W]$ to V_1 and $\perp[W]$ to V_2 , a concrete system described by AE is $E = \{X = V_1, Y = V_2\}$ and a substitution described by the annotation is $\sigma = \{V_1/f(g(U_1, U_2)), V_2/h(U_2)\}$. Again $E\sigma = \{X = f(g(U_1, U_2)), Y = h(U_2)\}$ gives the bindings of the reference variables.

Given a description $\langle AE_{in}, d_{in} \rangle$ and a set of equations E , abstract unification starts in a state $\langle E \cup AE_{in}, d_{in} \rangle$ and finishes in a state $\langle AE_{out}, d_{out} \rangle$ where AE_{out} is in an abstract solved form, i.e., a form which exhibits the relevant properties of the reference variables. Under this formalization, abstract unification consists of a sequence of rewrite steps. Each step rewrites a pair $\langle \mathcal{E}_{in}, d_{in} \rangle$ into a pair $\langle \mathcal{E}_{out}, d_{out} \rangle$ such that the following correctness criterion holds:

$$\langle \mathcal{E}_{in}, d_{in} \rangle \propto E_{in} \cup eqn(\sigma) \Rightarrow \exists E_{out}, \mu. \langle \mathcal{E}_{out}, d_{out} \rangle \propto E_{out} \cup eqn(\mu)$$

where $mgu(E_{in} \cup eqn(\sigma)) \equiv_V mgu(E_{out} \cup eqn(\mu))$.

Typically, some rewrite steps are *exact*. This is the case if for every $\langle E_{out}, \mu \rangle$ such that $\langle \mathcal{E}_{out}, d_{out} \rangle \propto E_{out} \cup eqn(\mu)$ there exist E_{in} and σ such that $\langle \mathcal{E}_{in}, d_{in} \rangle \propto E_{in} \cup eqn(\sigma)$ and $mgu(E_{out} \cup eqn(\mu)) \equiv_V mgu(E_{in} \cup eqn(\sigma))$. Typical examples of exact steps are steps which effect only the equation component, e.g. the steps which rewrite $\langle E, d \rangle$ to $\langle mgu(E), d \rangle$ in traditional analyses. These are called preunification steps in [42]. Proving their correctness is a rather trivial affair. In abstract equation systems, other steps are also in this category. In [8], steps leaving all representations of abstract variables (meta-terms in the paper) intact fall in this category.

The more challenging steps, which are the main subject of Section 7 are those which update the annotation component.

7 A Methodology for Developing Correct Analyses

An abstract unification algorithm is designed by mimicking a chosen set of equivalence preserving transitions on concrete equations. The precise transitions chosen depend on the abstract domain but in general include the rewrite rules of the classic Martelli-Montanari unification algorithm. Given a pair $\langle \varepsilon :: \mathcal{E}_{in}, d_{in} \rangle$ the abstract unification algorithm will typically aim to solve or reduce a single abstract equation ε , update the equations $\mathcal{E}_{in}$ and the description d_{in} and then carry on to select another abstract equation in the new system. The abstract

$$\begin{array}{ccc}
\varepsilon :: \mathcal{E}_{in}, d_{in} & \propto & e :: E_{in} \cup eqn(\sigma) \\
\downarrow & & \downarrow \\
\mathcal{E}_{out}, d_{out} & \propto & E_{out} \cup eqn(\mu)
\end{array}$$

Fig. 1. Correctness of Abstract Unification.

operation intends to compute a safe approximation of an equivalence preserving transformation (with respect to the reference domain V) from a corresponding $e :: E_{in} \cup eqn(\sigma)$ to $E_{out} \cup eqn(\mu)$. This situation is illustrated in Figure 1.

There are several candidates for such equivalence preserving transformations. A first commonly applied group reduces e to E' , possibly updating the equations of E_{in} , but does not affect $eqn(\sigma)$. An example of such a transformation, which also leaves E_{in} intact, is the peeling of an equation $f(t_1, \dots, t_n) = f(s_1, \dots, s_n)$ in the Martelli-Montanari algorithm. Another example is a variant of the Martelli-Montanari substitution rule. With $e \equiv x = t$, it applies the substitution x/t on E_{in} but not on $eqn(\sigma)$. Such transformations are typically applied to justify the exact steps mentioned in the previous section. A second group of commonly applied transformations solves the equation $e\sigma$ and applies the result on the remaining equations so that $E_{out} = E_{in} mgu(\{e\sigma\})$ and $\mu = \sigma mgu(\{e\sigma\})|V$. Often, concerns are separated in such a way that $E_{in} mgu(\{e\sigma\}) = E_{in}$. Also, $e\sigma$ typically has no reference variables and can be dropped (it is “absorbed” in $eqn(\mu)$). In the abstract algorithm one applies the result of “solving” ε under d_{in} to the equations in $\mathcal{E}_{out}$ and update d_{out} to describe all possible μ as specified above for all σ described by d_{in} .

Breaking up the abstract unification operator in a number of more simple operators transforming abstract equation systems makes abstract unification a lot easier to understand. As the operators are not necessarily commutative, abstract unification defined this way is not always confluent: the order of applying steps can influence the precision of an analysis. However, this need not reduce precision compared to complex one step operators as the order of applying operators can be controlled. The operations we use below depend only on the properties of the lhs. and rhs. of the selected equation. Other operators, taking the equation apart have to be applied first if properties of components are relevant.

We have observed that the design and the justification of abstract unification algorithms as well as the understanding of the underlying intuition crucially depend on the careful analysis of the relationship between the properties of σ , e , and μ reflected in the abstract domain. We distinguish between *definite* and *possible* properties.

Definite properties are properties such as freeness, groundness and covering information which if d_{out} claims them to hold, must indeed hold in μ . The abstract operation typically considers a case analysis on d_{in} and ε to guarantee that for any corresponding σ and e the property claimed about d_{out} indeed holds for $\mu = \sigma mgu(e\sigma)|V$.

Possible properties are properties such as sharing which if d_{out} claims them

to hold might hold in μ . Here it is the careful analysis of concrete unification which relates the assumed property of μ to the properties of σ and e . Since $\langle \varepsilon :: \mathcal{E}_{in}, d_{in} \rangle \propto e :: E_{in} \cup eqn(\sigma)$, the presence of these properties indicates the presence of certain abstract properties in ε, d_{in} . Correctness requires that the abstract operation is designed such that the property assumed in μ indeed shows up in d_{out} .

In the remainder of this section we will carefully unravel the relationships between σ , $e\sigma$, and μ that are the basis for showing correctness of challenging absorption steps in a number of well-known domains.

Set-Sharing Analysis The abstract domain used is $\wp(\wp(V))$. The initial assumption is $sharegroups(\{\sigma\}) \subseteq d_{in}$ and the unification problem is $s = t$ with $\text{Var}(\langle s, t \rangle) \subseteq V$. Let $\theta = mgu(\{s = t\}\sigma)$, and $\mu = \sigma\theta \upharpoonright V$. As stated above, it must be proven that, if $S \in sharegroups(\{\mu\})$, then $S \in d_{out}$. We borrow the basic structure of the original soundness proof as presented in [23].

The first part of the proof relates the existence of S in $sharegroups(\{\mu\})$ to the sharing groups of σ and the properties of s and t . Observe that $S \in sharegroups(\{\mu\})$ iff $\exists U. S = sharegroup(\mu, U) \neq \emptyset$. For each such variable U , two cases have to be considered:

1. $U \notin \text{Var}(\langle s\sigma, t\sigma \rangle)$. Since θ is relevant and idempotent, $U \notin \text{Var}(s\sigma\theta)(= \text{Var}(t\sigma\theta))$ and therefore, for $X \in V$, $U \in \text{Var}(X\sigma\theta) \Leftrightarrow U \in \text{Var}(X\sigma)$. Thus $S = sharegroup(\sigma, U)$, i.e., $S \in sharegroups(\{\sigma\})$. Also, since $U \notin \text{Var}(\langle s\sigma, t\sigma \rangle)$, an interesting property of S is: $S \cap \text{Var}(\langle s, t \rangle) = \emptyset$.
2. $U \in \text{Var}(\langle s\sigma, t\sigma \rangle)$ and since θ is relevant and idempotent $U \in \text{Var}(s\sigma\theta)(= \text{Var}(t\sigma\theta))$. The set $SS_s = \{Z \in \text{Var}(s\sigma) \mid U \in \text{Var}(Z\theta)\} \neq \emptyset$ (Figure 2) and similarly $SS_t = \{Z \in \text{Var}(t\sigma) \mid U \in \text{Var}(Z\theta)\} \neq \emptyset$. Since $\mu = \sigma\theta \upharpoonright V$, we have

$$S = \bigcup_{Z \in SS_s} sharegroup(\sigma, Z) \cup \bigcup_{Z \in SS_t} sharegroup(\sigma, Z).$$

Also the following property holds: $\forall Z \in SS_s. sharegroup(\sigma, Z) \cap \text{Var}(s) \neq \emptyset$, and $\forall Z \in SS_t. sharegroup(\sigma, Z) \cap \text{Var}(t) \neq \emptyset$.

Fig. 2. An occurrence of U in $s\sigma\theta$.

For a set-sharing unification algorithm to be sound, it is sufficient to capture both cases, that is, on the one hand to pass on sharing groups $S \in d_{in}$ satisfying the property $S \cap \text{Var}(\langle s, t \rangle) = \emptyset$ to d_{out} and on the other hand to include all unions of sets of sharing groups of which at least one has a non-empty intersection with $\text{Var}(s)$ and at least one a non-empty intersection with $\text{Var}(t)$. For the latter purpose, so-called *set-closure* operations are generally introduced which ensure all such unions are included in d_{out} , with the least possible other spurious unions of sharing groups.

Example 3. Let $V = \{X, Y, P, Q, R, T\}$, $d_{in} = \{\{X, R\}, \{Y\}, \{T, P\}, \{Q, X\}, \{Q\}\}$, $\sigma = \{X/f(W, W, R), Y/U, Q/g(W, B), T/P\}$, and let $s = t$ be the equation $X = f(Y, T, P)$. Then $d_{in} \propto \sigma$ and the unification $s\sigma = t\sigma$ to be abstracted is $f(W, W, R) = f(U, P, P)$. A most general unifier is $\theta = \{W/U, R/U, P/U\}$. The reader can easily check that $sharegroup(\mu, U) = \{X, R, Q, Y, T, P\}$, which is obtained as the union of four sharing groups from d_{in} . On the one hand, as $SS_s = \{W, R\}$, we need $sharegroup(\sigma, W) = \{Q, X\}$ and $sharegroup(\sigma, R) = \{X, R\}$, and on the other hand, as $SS_t = \{U, P\}$, we need $sharegroup(\sigma, U) = \{Y\}$ and $sharegroup(\sigma, P) = \{T, P\}$.

If $ground_\sigma(s)$ (or $ground_\sigma(t)$), then $\text{Var}(s\sigma\theta) (= \text{Var}(t\sigma\theta)) = \emptyset$. In the set-sharing domain, this can be detected if no variable from s (or t) occurs in d_{in} . The abstract algorithm then only needs to capture case 1. above (e.g., $d_{out} = \{S \in d_{in} \mid S \cap \text{Var}(\langle s, t \rangle) = \emptyset\}$). Informally, the groundness information of d_{in} is preserved, the groundness of the variables of s is propagated to the variables of t (or vice versa) and through the (limited) covering information encoded in d_{in} also to some other variables in V .

Further refinements are possible if the set-sharing domain is combined with a mode domain providing linearity and/or freeness information. For instance, by taking into account the following special cases:

- For $U \in \text{Var}(s\sigma\theta)$, if $linear_\sigma(s)$ (resp. $linear_\sigma(t)$) and $indep_\sigma(s, t)$, then SS_t (resp. SS_s) is a singleton set.
- For $U \in \text{Var}(s\sigma\theta)$, if $free_\sigma(s)$ or $free_\sigma(t)$ or $linear_\sigma(\langle s, t \rangle)$, then both SS_s and SS_t are singleton sets. (Note: $linear_\sigma(\langle s, t \rangle) \rightarrow indep_\sigma(s, t)$.)

Based on these observations, the set-closure operations can be optimized [7, 25, 27, 40].

Example 4. Let $V = \{X, P, Q, R\}$ and assume d_{in} consists of both a set-sharing component $\{\{X\}, \{P, R\}, \{P, Q\}\}$ and a mode component $\{X/free, P/any, Q/any, R/free\}$. Then $\sigma = \{P/f(R, U), Q/U\}$ is described by d_{in} . Let $s = t$ be the equation $X = f(P, R)$. The unification $s\sigma = t\sigma$ to be abstracted is $X = f(f(R, U), R)$. A most general unifier is $\theta = \{X/f(f(R, U), R)\}$. For U both $SS_s = \{X\}$ and $SS_t = \{U\}$ are singleton sets and $sharegroup(\mu, U) = \{X, P, Q\}$ is obtained as the union of $sharegroup(\sigma, X) = \{X\}$ and $sharegroup(\sigma, U) = \{P, Q\}$.

The same reasoning has been applied in the domain of abstract equation systems [38] where s and t are terms built from abstract variables and d_{in} expresses the sharing between abstract variables (see Section 8).

Pair-Sharing Analysis The domain used is $\wp(V \times V)$, most often in combination with a groundness component in $\wp(V)$. Let $d_{in} = \langle d_{in}^{share}, d_{in}^{gr} \rangle$. The sharing component d_{in}^{share} is in fact required to be a symmetrical binary relation satisfying the property $d_{in}^{share} \cap (d_{in}^{gr} \times V) = \emptyset$. Again, the unification problem $s = t$ is such that $\text{Var}(\langle s, t \rangle) \subseteq V$. The initial assumption is $\sigma \in \gamma(d_{in})$, i.e. $\forall X, Y \in V. X \neq Y \& share_{\sigma}(X, Y) \rightarrow (X, Y) \in d_{in}^{share}$ and (following the approach of [42]) $\forall X \in V. \neg linear_{\sigma}(X) \rightarrow (X, X) \in d_{in}^{share}$. It must be proven that, if $X \neq Y \& share_{\mu}(X, Y)$, then $(X, Y) \in d_{out}^{share}$. We borrow the structure of a soundness proof presented in [9].

First note that $share_{\mu}(X, Y)$ iff $share_{\theta}(X\sigma, Y\sigma)$. The sharing between $X\sigma$ and $Y\sigma$ under the substitution θ is related to the sharing between X, Y , and the variables of s and t under the initial substitution σ , and can be expressed as follows:

$$share_{\theta}(X\sigma, Y\sigma) \rightarrow share_{\sigma}(X, Y) \vee \begin{cases} \exists Z_X, Z_Y \in \text{Var}(\langle s, t \rangle). \\ share_{\sigma}(X, Z_X) \& share_{\sigma}(Y, Z_Y) \\ \& share_{\theta}(Z_X\sigma, Z_Y\sigma) \end{cases}$$

Note that cases with $X = Z_X$ and/or $Y = Z_Y$ are captured by the above formula. For a pair-sharing unification algorithm to be sound, it is sufficient to capture both parts of the consequent. For the first disjunct, it is sufficient to pass on the pairs $(X, Y) \in d_{in}^{share}$ to d_{out}^{share} . For the second disjunct, abstract unification algorithms generally introduce an operation that computes an appropriate subset of the transitive closure of a binary relation [9, 41]. For example,

$$Compose(s = t, d_{in}^{share}) = d_{in}^{share} \cup \left\{ (X, Y) \left| \begin{array}{l} \exists Z_X, Z_Y \in \text{Var}(\langle s, t \rangle). \\ ((X, Z_X) \in d_{in}^{share} \vee X = Z_X) \& \\ ((Y, Z_Y) \in d_{in}^{share} \vee Y = Z_Y) \& \\ (Z_X, Z_Y) \in NewShare(s, t) \end{array} \right. \right\}$$

Here, $NewShare(s, t)$ must capture the sharing between the variables of s and t under the substitution μ . For instance, defining $NewShare(s, t) = \text{Var}(\langle s, t \rangle) \times \text{Var}(\langle s, t \rangle)$ yields a sound abstract unification operation. However, as in the set-sharing case, the algorithm can be refined when using definite groundness and/or linearity information.

For instance, if $ground_{\sigma}(s)$ (or $ground_{\sigma}(t)$) is derivable from the groundness component d_{in}^{gr} , the abstract algorithm only needs to capture the first part of the consequent above, e.g., defining $d_{out}^{share} = \{(X, Y) \in d_{in}^{share} \mid X, Y \notin \text{Var}(\langle s, t \rangle)\}$ and $d_{out}^{gr} = d_{in}^{gr} \cup \text{Var}(\langle s, t \rangle)$. Informally, the definite groundness information of d_{in} is preserved and the groundness of the variables of s is propagated to the variables of t (or vice versa). Since no covering information is encoded in the pair-sharing domain, no further propagation of groundness is possible.

Special properties exploiting linearity enable optimizations in the computation of $NewShare(s, t)$. For instance,

- If $linear_\sigma(\langle s, t \rangle)$, $X, Y \in \text{Var}(\langle s\sigma, t\sigma \rangle)$, and $X \neq Y$, then $share_\theta(X, Y) \rightarrow (X \in \text{Var}(s\sigma) \leftrightarrow Y \in \text{Var}(t\sigma))$.
Informally, under the conditions mentioned, X and Y occur on opposite sides of the equation $s\sigma = t\sigma$.
- If $linear_\sigma(s)$, $indep_\sigma(\langle s, t \rangle)$, $X, Y \in \text{Var}(\langle s\sigma, t\sigma \rangle)$, and $X \neq Y$, then $share_\theta(X, Y) \rightarrow (X \in \text{Var}(t\sigma) \rightarrow Y \in \text{Var}(s\sigma))$.

The original pair-sharing analysis [9, 42] incorporated these optimizations.

Example 5. Let $V = \{X, Y, Z, W, R\}$, $d_{in}^{share} = \{(W, X), (R, X)\}$ (for brevity, we use a condensed representation for the symmetrically closed sharing relation), and $d_{in}^{gr} = \{\}$. Then $\sigma = \{X/f(W, R), Z/g(U)\}$ is described by d_{in} . Let $s = t$ be the equation $X = f(Y, Z)$. The unification $s\sigma = t\sigma$ to be abstracted is $f(W, R) = f(Y, g(U))$. A most general unifier is $\theta = \{W/Y, R/g(U)\}$. Without using linearity information, a rough $NewShare(s, t)$ is obtained as $\{(X, X), (Y, Y), (Z, Z), (X, Y), (Y, Z), (X, Z)\}$. Using the linearity and independence information in d_{in} , an optimized $NewShare(s, t)$ is $\{(X, Y), (X, Z)\}$. In the latter case, using $Compose$ defined above, the reader can verify that d_{out}^{share} is $\{(W, X), (R, X), (X, Y), (X, Z), (W, Y), (W, Z), (R, Y), (R, Z)\}$. The properties $indep_\mu(W, R)$ and $indep_\mu(Y, Z)$ are captured in the latter case, but would not when using the less optimized definition of $NewShare(s, t)$.

Groundness Analysis Groundness information is derived in both the pair- and set-sharing analyses, but can also be derived on its own in a simple mode domain. The general proof obligation can be formulated as follows: if d_{out} constrains a variable $X \in V$ to be bound to a ground term, then $ground_\mu(X)$ must hold. The first part of a soundness proof relates the groundness information present in d_{out} to the groundness information in d_{in} and particular properties of $s = t$. The second part of the proof then uses $d_{in} \propto \sigma$ and a few obvious properties about the way in which groundness is propagated.

- $\forall X \in V. ground_\sigma(X) \rightarrow ground_\mu(X)$;
- $ground_\sigma(s) \rightarrow \forall Z \in \text{Var}(t). ground_\mu(Z)$;
(and a symmetrical property if $ground_\sigma(t)$).

In the case where (depth-bounded) structural information is derived [16, 38], groundness of subterms may be expressible and can be proven correct using a similar proof strategy.

Freeness and Linearity Analyses Abstract unification algorithms deriving definite freeness and/or linearity information often consist of a case analysis that can be proven correct using a similar scheme as described above. In essence, the proof is based on a set of sufficient conditions for freeness/linearity to be preserved under the application of a most general unifier.

For freeness, for instance, the following properties are intuitively justifiable. For $X \in V$,

- $free_\sigma(X) \ \& \ free_\sigma(s) \ \& \ free_\sigma(t) \rightarrow free_\mu(X)$;
(informally, if two variables are being unified, no other variable gets bound to a non-variable term)
- $free_\sigma(X) \ \& \ indep_\sigma(X, \langle s, t \rangle) \rightarrow free_\mu(X)$;
(informally, free variables that are independent of the terms being unified do not get instantiated to a non-variable term).
- $free_\sigma(X) \ \& \ free_\sigma(s) \ \& \ indep_\sigma(X, s) \rightarrow free_\mu(X)$
(and symmetrical for the term t);
(informally, if one of the terms being unified is a free variable, then other variables independent of that term do not get bound to a non-variable term).

Since these conditions involve independence information, a freeness analysis usually is combined with a sharing analysis in order for non-trivial freeness information to be derivable (e.g., [14, 26, 40]).

The same holds for a linearity analysis. Sufficient conditions for linearity to be derivable are: For $X \in V$,

- $ground_\mu(X) \rightarrow linear_\mu(X)$;
- $linear_\sigma(X) \ \& \ indep_\sigma(X, \langle s, t \rangle) \rightarrow linear_\mu(X)$;
- $linear_\sigma(X) \ \& \ linear_\sigma(s) \ \& \ indep_\sigma(X, s) \rightarrow linear_\mu(X)$
(and symmetrical for the term t).

Deriving linearity information is particularly useful to enhance set-sharing analysis [7, 25, 27]. It was already intrinsically integrated in the pair-sharing analysis presented by Søndergaard [42].

Covering Analysis The best known abstract domain for deriving covering information is Prop [4, 30, 34, 43], although set-sharing [23] and some other approaches [14] also used this kind of information, primarily to improve groundness analysis.

Useful properties for justifying covering analyses are:

- $\forall X \in \text{Var}(s) : covers_\mu(\text{Var}(t), X)$; and symmetrically
- $\forall X \in \text{Var}(t) : covers_\mu(\text{Var}(s), X)$;
- For $S \subseteq V$, $X \in V$: $covers_\sigma(S, X) \rightarrow covers_\mu(S, X)$;
- For $S, T \subseteq V$, $X, Y \in V$:
 $covers_\mu(T, X) \ \& \ covers_\mu(\{X\} \cup S, Y) \rightarrow covers_\mu(T \cup S, Y)$.

8 Abstract Equation Systems

The formulation of abstract unification above and the properties of concrete substitutions are basic means that facilitate the development and justification of analyses that combine the derivation of mode, variable dependencies, and structural information [7, 16]. In particular, the representation has proven useful in the design of correct abstract unification algorithms involving freeness information [8]. By way of example, we illustrate the domain of abstract equation

systems [7, 38], which augments the set-sharing domain with a mode component and depth-bounded structural information.

In this approach *abstract* variables are introduced to represent (concrete) terms whose internal structure is hidden by abstraction. We denote by AVar the denumerable set of abstract variables (disjoint from Var). Abstract variables are represented using $@, @_i$, etc. to distinguish them from concrete variables. The set of abstract terms, denoted ATerm , is constructed using function symbols from Σ and variables from AVar . The set of abstract variables occurring in a syntactic object s is denoted $\text{AVar}(s)$.

An abstract equation system AE , for reference domain V , is a triplet $\langle \mathcal{E}; \iota; \Delta \rangle$ such that $\mathcal{E}$ is a finite set of equations over $\text{Term} \cup \text{ATerm}$ with $\text{Var}(\mathcal{E}) \subseteq V$, $\Delta \subseteq \wp(\text{AVar}(\mathcal{E}))$ is a set-sharing component, and $\iota : \text{AVar}(\mathcal{E}) \rightarrow \{f, \ell, g, a\}$ is an annotation mapping which indicates for each abstract variable if it corresponds to a *free*, *linear*, *ground* or an *arbitrary* concrete term. The set of annotations, $\{f, \ell, g, a\}$ is partially ordered such that $f \preceq \ell \preceq a$ and $g \preceq \ell \preceq a$, corresponding to the intuition that free and ground terms are also linear and that free, linear and ground terms are also arbitrary terms. The abstract equation system is said to be in *solution solved form* if $\mathcal{E}$ contains for each $X \in V$ one equation of the form $X = t$ with $t \in \text{ATerm}$.

Informally, the description relation is defined via an abstract-variable replacement which is a mapping that assigns a concrete variable outside the reference domain V to each abstract variable, and a substitution σ having this set of newly introduced variables V' as domain and all range variables outside both V and V' ; moreover, σ should be such that the mode and possible sharing restrictions are met.

Example 6. Let $V = \{X, Y, Z, U\}$ and

$$AE = \left\langle \left\{ \begin{array}{l} X = @_4, Y = @_2, \\ Z = h(@_3, @_1), U = @_1 \end{array} \right\}; \left\{ \begin{array}{l} @_1/f, @_2/\ell, \\ @_3/\ell, @_4/a \end{array} \right\}; \left\{ \begin{array}{l} \{ @_2, @_3 \}, \{ @_1 \}, \\ \{ @_1, @_4 \}, \{ @_4 \} \end{array} \right\} \right\rangle.$$

This abstract equation system (in solution solved form) represents many concrete equation systems (and corresponding substitutions). For example,

$$E = \left\{ \begin{array}{l} X = V_4, Y = V_2, \\ Z = h(V_3, V_1), U = V_1 \end{array} \right\},$$

is represented via the abstract-variable replacement

$$\theta : @_1 \mapsto V_1, @_2 \mapsto V_2, @_3 \mapsto V_3, @_4 \mapsto V_4,$$

and a substitution described by the annotations and sharing component is

$$\sigma : V_1 \mapsto P, V_2 \mapsto f(W), V_3 \mapsto h(W, a), V_4 \mapsto g(P, R, P).$$

The bindings for the reference variables are then obtained from

$$E\sigma = \left\{ \begin{array}{l} X = g(P, R, P), Y = f(W), \\ Z = h(h(W, a), P), U = P \end{array} \right\}.$$

The abstract unification algorithm consists of a set of rewrite rules that reduce an abstract equation system to solution solved form which clearly exhibits the desired freeness, groundness, linearity, and sharing properties. We illustrate some rewriting steps for reducing a unification problem $s = t, AE_{in}$ to a data description AE_{out} . Below, the annotation mapping ι is represented implicitly by annotating each occurrence of an abstract variable $@$ with a superscript indicating the value of $\iota(@)$.

Example 7. Let $V = \{X, Y, Z, U\}$, consider an initial data description

$$AE_{in} = \left\langle \begin{cases} X = @_4^f, \\ Y = @_2^\ell, \\ Z = h(@_3^\ell), \\ U = @_1^f \end{cases}, \left\{ \left\{ @_2^\ell, @_3^\ell \right\}, \left\{ @_1^f \right\} \right\} \right\rangle,$$

and let the equation $s = t$ to be solved be $U = f(Y, Z)$.

In the *preunification phase*, rewriting steps similar to the Martelli-Montanari unification algorithm are applied. The rewriting rules preserve structural information if possible. For instance, the equation component $\mathcal{E}$ of AE_{in} is first applied as a kind of substitution to the equation $U = f(Y, Z)$:

$$\left\langle \left\{ @_1^f = f(@_2^\ell, h(@_3^\ell)) \right\}, \left\langle \begin{cases} X = @_4^f, \\ Y = @_2^\ell, \\ Z = h(@_3^\ell), \\ U = @_1^f \end{cases}, \left\{ \left\{ @_2^\ell, @_3^\ell \right\}, \left\{ @_1^f \right\} \right\} \right\rangle \right\rangle.$$

The abstract equation is then used as a substitution $\{@_1^f / f(@_2^\ell, h(@_3^\ell))\}$ on $\mathcal{E}$, thereby propagating structural information to the variable $U \in V$.

$$\left\langle \left\{ @_1^f = f(@_2^\ell, h(@_3^\ell)) \right\}, \left\langle \begin{cases} X = @_4^f, \\ Y = @_2^\ell, \\ Z = h(@_3^\ell), \\ U = f(@_2^\ell, h(@_3^\ell)) \end{cases}, \left\{ \left\{ @_2^\ell, @_3^\ell \right\}, \left\{ @_1^f \right\} \right\} \right\rangle \right\rangle.$$

The sharing and annotation components are only affected in the *absorption phase*. Since freeness and linearity information is available, the optimized set-sharing algorithm can be mimicked here, yielding the final data description:

$$AE_{out} = \left\langle \begin{cases} X = @_4^a, \\ Y = @_2^\ell, \\ Z = h(@_3^\ell), \\ U = f(@_2^\ell, h(@_3^\ell)) \end{cases}, \left\{ \left\{ @_2^\ell, @_3^\ell, @_4^a \right\}, \left\{ @_2^\ell, @_3^\ell \right\}, \left\{ @_4^a \right\} \right\} \right\rangle.$$

The freeness of the abstract variable $@_4$ is not preserved in the absorption step since $@_4$ is possibly dependent (see sharing group $\{@_1^f, @_4^f\}$) on the left-hand side $@_1$ of the equation being absorbed, the right-hand side of which is possibly non-linear (see sharing group $\{@_2^\ell, @_3^\ell\}$).

9 Conclusion

We reviewed a number of analyses using (abstract) equation systems as a common basis to decompose the key abstract operation, namely, unification in the abstract domain, into more simple sub-problems. A modular approach results

which eases the design and justification of the analysis. The essential remaining step in developing a correct abstraction is a careful analysis of the underlying concrete unification with respect to the properties of interest. We summarized a collection of properties that have been the basis of several abstract domains described in the literature.

Acknowledgment

We have tried in this paper to present the work of many different authors within a representation of abstract equation systems. We feel that this gives a somewhat unifying view to a field in which many people have made contributions. Discussions and collaborations with many of these authors over the past few years have contributed immensely to this work. We thank them all.

References

1. S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Series in Computers and their Applications. Ellis Horwood, Chichester, 1987.
2. K. R. Apt. Logic programming. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 493–574. Elsevier, 1990.
3. K. R. Apt and A. Pellegrini. On the occur-check-free Prolog programs. *ACM Trans. Prog. Lang. Syst.*, 16(3):687–726, 1994.
4. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In B. Le Charlier, editor, *Proceedings of the first International Static Analysis Symposium, SAS'94*, pages 266–280. LNCS 864, Springer-Verlag, Sept. 1994.
5. R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Trans. Prog. Lang. Syst.*, 15(1):133–181, Jan. 1993.
6. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10(2):91–124, 1991.
7. 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*, LNCS 724, pages 153–164, Padova, Italy, Sept. 1993. Springer-Verlag.
8. 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. Revised version to appear in J. of Logic Programming.
9. M. Codish, D. Dams, and E. Yardeni. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 79–93, Paris, France, 1991. The MIT Press, Cambridge, Mass.
10. M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science*, 124(1):93–126, Feb. 1994.

11. M. Codish and B. Dømoen. Analysing logic programs using “prop”-ositional logic programs and a magic wand. *The Journal of Logic Programming*. (to appear).
12. M. Codish, M. Falaschi, and K. Marriott. Suspension analysis for concurrent logic programs. *ACM Trans. Prog. Lang. Syst.*, 16(3):649–686, May 1994.
13. M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. In *Proceedings of PEPM'93, the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 194–205, Copenhagen, June 1993. Revised version to appear in *ACM Trans. Prog. Lang. Syst.*
14. A. Cortesi and G. Filé. Abstract interpretation of logic programs: An abstract domain for groundness, sharing, freeness and compoundness analysis. In P. Hudak and N. D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 52–61, 1991.
15. A. Cortesi, G. Filé, and W. Winsborough. Comparison of abstract interpretations. In *Proceedings of ICALP'92*, pages 521–532. LNCS Vol. 623, 1992.
16. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming. In *Proceedings of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 227–239, Portland, Oregon, 1994. ACM Press.
17. 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, Los Angeles, 1977.
18. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 and 3):103–179, 1992.
19. S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Trans. Prog. Lang. Syst.*, 11(3):418–450, 1989.
20. T. W. Getzinger. The costs and benefits of abstract interpretation-driven Prolog optimization. In B. Le Charlier, editor, *Proceedings of the First International Static Analysis Symposium, SAS'94*, LNCS 864, pages 1–25. Springer-Verlag, 1994.
21. M. V. Hermenegildo and F. Rossi. Strict and non-strict independent AND-parallelism in logic programs: Correctness, efficiency, and compile-time conditions. *The Journal of Logic Programming*, 22(1):1–45, Jan. 1995.
22. D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 154–165, Cleveland, Ohio, USA, 1989.
23. D. Jacobs and A. Langen. Static analysis of logic programs for independent AND-parallelism. *The Journal of Logic Programming*, 13(2 & 3):291–314, 1992.
24. N. D. Jones and H. Søndergaard. A semantic-based framework for the abstract interpretation of Prolog. In Abramsky and Hankin [1], pages 123–142.
25. A. King. A synergistic analysis for sharing and groundness which traces linearity. In D. Sannella, editor, *Proceedings of Fifth European Symposium on Programming Languages and Systems – ESOP'94*, pages 363–378, Edinburgh, 1994. Springer-Verlag.
26. 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.
27. A. Langen. *Advanced Techniques for Approximating Variable Aliasing in Logic Programs*. Ph. D. thesis, University of Southern California, Mar. 1991.

28. 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 Publishers Inc., Los Altos, 1988.
29. B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of Prolog. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 750–764, Washington D.C., 1992. MIT Press, Cambridge.
30. B. Le Charlier and P. Van Hentenryck. Groundness analysis for Prolog: Implementation and evaluation of the domain Prop. In *Proceedings of PEPM'93, the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 99–110, Copenhagen, June 1993.
31. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Trans. Prog. Lang. Syst.*, 16(1):35–101, Jan. 1994.
32. J. W. Lloyd. *Foundations of Logic Programming*. Springer Series : Symbolic Computation - Artificial Intelligence. Springer-Verlag, second, extended edition, 1987.
33. K. Marriott. Frameworks for abstract interpretation. *Acta Informatica*, 30(2):103–129, 1993.
34. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Lett. Prog. Lang. Syst.*, 2(1-4):181–196, 1993.
35. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, Apr. 1982.
36. C. S. Mellish. The automatic generation of mode declarations for Prolog programs. DAI Research Paper 163, Dept. of Artificial Intelligence, Univ. of Edinburgh, Aug. 1981.
37. C. S. Mellish. Abstract interpretation of Prolog programs. In Abramsky and Hankin [1], pages 181–198.
38. 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*, Kanagawa, Japan, 1995. MIT Press. To appear.
39. K. Muthukumar and M. Hermenegildo. Determination of variable dependence information through abstract interpretation. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 166–188, Cleveland, Ohio, USA, 1989.
40. K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 49–63, Paris, France, 1991. The MIT Press.
41. D. A. Plaisted. The occur-check problem in Prolog. *J. New Generation Computing*, 2(4):309–322, 1984. Also in: *Proceedings of the International Symposium on Logic Programming*, pages 272–280, Atlantic City, 1984. IEEE Computer Society Press.
42. H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *ESOP'86 Proceedings European Symposium on Programming*, LNCS 213, pages 327–338. Springer-Verlag, N.Y., 1986.
43. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain Prop. *The Journal of Logic Programming*, 23(3):237–278, 1995.

This article was processed using the L^AT_EX macro package with LLNCS style