A Synergistic Analysis for Sharing and Groundness which traces Linearity

Andy King

Department of Electronics and Computer Science,**
The University of Southampton, Southampton, S09 5NH, UK.

Abstract. Accurate variable sharing information is crucial both in the automatic parallelisation and in the optimisation of sequential logic programs. Analysis for possible variable sharing is thus an important topic in logic programming and many analyses have been proposed for inferring dependencies between the variables of a program, for instance, by combining domains and analyses. This paper develops the combined domain theme by explaining how term structure, and in particular linearity, can be represented in a sharing group format. This enables aliasing behaviour to be more precisely captured; groundness information to be more accurately propagated; and in addition, refines the tracking and application of linearity. In practical terms, this permits aliasing and groundness to be inferred to a higher degree of accuracy than in previous proposals and also can speed up the analysis itself. Correctness is formally proven.

1 Introduction

Abstract interpretation for possible sharing is an important topic of logic programming. Sharing (or aliasing) analysis conventionally infers which program variables are definitely grounded and which variables can never be bound to terms containing a common variable. Applications of sharing analysis are numerous and include: the sound removal of the occur-check [22]; optimisation of backtracking [3]; the specialisation of unification [24]; and the elimination of costly checks in independent and-parallelism [20, 14, 21]. Early proposals for sharing analysis include [25, 10, 19].

This paper is concerned with a semantic basis for sharing analysis, and in particular, the justification of a high precision abstract unification algorithm. Following the approach of abstract interpretation [8], the abstract unification algorithm (the abstract operation) essentially mimics unification (the concrete operation) by finitely representing substitutions (the concrete data) with sharing abstractions (the abstract data). The accuracy of the analysis depends, in part, on the substitution properties that the sharing abstractions capture. Sharing abstractions usually capture groundness and aliasing information, and indeed, accurate analyses are often good at groundness propagation [14, 21]. A knowledge of groundness can improve sharing and *vice versa*. A synergistic relationship

** New address: The Computing Laboratory, The University of Kent, Canterbury, CT2 7LX, UK.

also exists between sharing and type analysis. Type analysis deduces structural properties of aggregate data. By keeping track of type information, that is inferring structural properties of substitutions, it is possible to infer more accurate sharing information. Conversely, more accurate type information can be deduced if sharing is traced.

Type information is often applied by combining sharing and freeness analysis [20, 7, 23] or by tracing linearity [22, 5]. Freeness information differentiates between a free variable, a variable which is definitely not bound to non-variable term; and a non-free variable, a variable which is possibly bound to a non-variable term. Freeness information is useful in its own right, in fact it is essential in the detection of non-strict and-parallelism [13]. A more general notion than freeness is linearity [22, 5]. Linearity relates to the number of times a variable occurs in a term. A term is linear if it definitely does not contain multiple occurrences of a variable; otherwise it is non-linear. Without exploiting linearity (or freeness), analyses have to assume that aliasing is transitive [5]. The significance of linearity is that the unification of linear terms only yields restricted forms of aliasing. Thus, if terms can be inferred to be linear, worst case aliasing need not be assumed in an analysis.

Sharing analyses can be used in isolation, but an increasing trend is to combine domains and analyses to improve accuracy [6]. For example, the pair-sharing domain of Søndergaard [22, 5], tracks linearity but is not so precise at propagating groundness information. Conversely, sharing group domains [14, 21] accurately characterise groundness but do not exploit linearity. The rationale behind [6], therefore, is to run multiple analyses in lock step. At each step, the sharing information from different analyses is compared and used to improve the precision. For instance, the linearity of the Søndergaard domain [22, 5] can be used to prune out spurious aliasing in the sharing group analysis [14, 21]; and the groundness information of the Jacobs and Langen domain can be used to remove redundant aliasing in the Søndergaard analysis.

This paper develops the combined domain theme by explaining how the linearity of the the Søndergaard domain [22, 5] can be represented in the sharing group format of the Jacobs and Langen domain [14, 21]. This enables both aliasing behaviour to be precisely captured, and groundness information to be accurately propagated, in a single coherent domain and analysis. This is not an exercise in aesthetics but has a number of important and practical implications:

1. By embedding linearity into sharing groups, the classic notion of linearity [22, 5] can be refined. Specifically, if a variable is bound to a non-linear term, it is still possible to differentiate between which variables of the term occur multiply in the term and which variables occur singly in the term. Put another way, the abstraction proposed in this paper records why a variable binding is potentially non-linear, rather than merely indicating that it is possibly non-linear. Previously, the variable would simply be categorised as non-linear, and worst-case aliasing assumed. The refined notion of linearity permits more accurate aliasing information to be squeezed out of the analysis. This can, in turn, potentially identify more opportunities for parallelism

and optimisation.

2. Tracking aliasing more accurately can also improve the efficiency of the analysis [6]. Possible aliases are recorded and manipulated in a data structure formed from sharing groups. As the set of possible aliases is inferred more accurately, so the set becomes smaller, and thus the number of sharing groups is reduced. The size of the data structures used in the analysis are therefore pruned, and consequently, analysis can proceed more quickly.
   Moreover, the sharing abstractions defined in this paper are described in terms of a single domain and manipulated by a single analysis. This is significant because, unlike the multiple analyses approach [6], it avoids the duplication of abstract interpretation machinery and therefore simplifies the analysis. In practical terms, this is likely to further speedup the analysis [12]. Furthermore, the closure under union operation implicit in the analyses of [14, 21] has exponential time- and space-complexity in the number of sharing groups. It is therefore important to limit its use. In this paper, an analog of closure under union operation is employed, but is only applied very conservatively to a restricted subset of the set of sharing groups. This is also likely to contribute to faster analysis.
3. Errors and omissions have been reported [4, 9] in some of the more recent proposals for improving sharing analysis with type information [20, 7, 23]. Although the problems relate to unusual or rare cases, and typically the analyses can be corrected, these highlight that analyses are often sophisticated, subtle and difficult to get right. Thus, formal proof of correctness is useful, indeed necessary, to instill confidence. For the analysis described in this paper, safety has been formally proved. In more pragmatic terms this means that the implementor can trust the results given by the analysis.

The exposition is structured as follows. Section 2 describes the notation and preliminary definitions which will be used throughout. Also, linearity is formally introduced and its significance for aliasing is explained. In section 3, the focus is on abstracting data. A novel abstraction for substitutions is proposed which elegantly and expressively captures both linear and sharing properties of substitutions. In section 4, the emphasis changes to abstracting operations. Abstract analogs for renaming, unification, composition and restriction are defined in terms of an abstract $unify$ operator [14]. An abstract unification algorithm is precisely and succinctly defined which, in turn, describes an abstract analog of $unify$. (Once an abstract $unify$ operator is specified and proved safe, a complete and correct abstract interpreter is practically defined by virtue of existing abstract interpretation frameworks [1, 17, 21].) Finally, sections 5 and 6 present the related work and the concluding discussion. For reasons of brevity and continuity, proofs are not included in the paper, but can be found in [15].

2 Notation and preliminaries

To introduce the analysis some notation and preliminary definitions are required. The reader is assumed to be familiar with the standard constructs used in logic programming [18] such as a universe of all variables $(u, v \in) Uvar$; the set of terms $(t \in) Term$ formed from the set of functors $(f, g, h \in) Func$ (of the first-order language underlying the program); and the set of program atoms $Atom$. It is convenient to denote $f(t_1, \ldots, t_n)$ by τ_n and $f'(t'_1, \ldots, t'_n)$ by τ'_n. Also let $\tau_0 = f$ and $\tau'_0 = f'$. Let $Pvar$ denote a finite set of program variables - the variables that are in the text of the program; and let $var(o)$ denote the set of variables in a syntactic object o.

2.1 Substitutions

A substitution ϕ is a total mapping $\phi : Uvar \rightarrow Term$ such that its domain $dom(\phi) = \{u \in Uvar \mid \phi(u) \neq u\}$ is finite. The application of a substitution ϕ to a variable u is denoted by $\phi(u)$. Thus the codomain is give by $cod(\phi) = \cup_{u \in dom(\phi)} var(\phi(u))$. A substitution ϕ is sometimes represented as a finite set of variable and term pairs $\{u \mapsto \phi(u) \mid u \in dom(\phi)\}$. The identity mapping on $Uvar$ is called the empty substitution and is denoted by ϵ. Substitutions, sets of substitutions, and the set of substitutions are denoted by lower-case Greek letters, upper-case Greek letters, and $Subst$.

Substitutions are extended in the usual way from variables to functions, from functions to terms, and from terms to atoms. The restriction of a substitution ϕ to a set of variables $U \subseteq Uvar$ and the composition of two substitutions ϕ and φ, are denoted by $\phi \upharpoonright U$ and $\phi \circ \varphi$ respectively, and defined so that $(\phi \circ \varphi)(u) = \phi(\varphi(u))$. The preorder $Subst$ ($\sqsubseteq$), ϕ is more general than φ, is defined by: $\phi \sqsubseteq \varphi$ if and only if there exists a substitution $\psi \in Subst$ such that $\varphi = \psi \circ \phi$. The preorder induces an equivalence relation $\approx$ on $Subst$, that is: $\phi \approx \varphi$ if and only if $\phi \sqsubseteq \varphi$ and $\varphi \sqsubseteq \phi$. The equivalence relation $\approx$ identifies substitutions with consistently renamed codomain variables which, in turn, factors $Subst$ to give the poset $Subst/\approx$ ($\sqsubseteq$) defined by: $[\phi]_\approx \sqsubseteq [\varphi]_\approx$ if and only if $\phi \sqsubseteq \varphi$.

2.2 Equations and most general unifiers

An equation is an equality constraint of the form $a = b$ where a and b are terms or atoms. Let $(e \in) Eqn$ denote the set of finite sets of equations. The equation set $\{e\} \cup E$, following [5], is abbreviated by $e : E$. The set of most general unifiers of E, $mgu(E)$, is defined operationally [14] in terms of a predicate mgu. The predicate $mgu(E, \phi)$ which is true if ϕ is a most general unifier of E.

Definition 1 ***mgu*.** The set of most general unifiers $mgu(E) \in \wp(Subst)$ is defined by: $mgu(E) = \{\phi \mid mgu(E, \phi)\}$ where

$$\begin{array}{rl}
mgu(\emptyset, \epsilon) & \\
mgu(v = v' : E, \zeta) & \text{if } mgu(E, \zeta) \quad \wedge v \equiv v' \\
mgu(v = v' : E, \zeta \circ \eta) & \text{if } mgu(\eta(E), \zeta) \wedge v \not\equiv v' \wedge \eta = \{v \mapsto v'\} \\
mgu(v = v' : E, \zeta \circ \eta) & \text{if } mgu(\eta(E), \zeta) \wedge v \not\equiv v' \wedge \eta = \{v' \mapsto v\} \\
mgu(v = \tau_n : E, \zeta \circ \eta) & \text{if } mgu(\eta(E), \zeta) \wedge v \notin var(\tau_n) \wedge \eta = \{v \mapsto \tau_n\} \\
mgu(\tau_n = v : E, \zeta \circ \eta) & \text{if } mgu(\eta(E), \zeta) \wedge v \notin var(\tau_n) \wedge \eta = \{v \mapsto \tau_n\} \\
mgu(\tau_n = \tau'_n : E, \zeta) & \text{if } mgu(t_1 = t'_1 : \ldots : t_n = t'_n : E, \zeta) \wedge f \equiv f'
\end{array}$$

By induction it follows that $dom(\phi) \cap cod(\phi) = \emptyset$ if $\phi \in mgu(E)$, or put another way, that the most general unifiers are idempotent [16].

Following [14], the semantics of a logic program is formulated in terms of a single $unify$ operator. To construct $unify$, and specifically to rename apart program variables, an invertible substitution [16], Υ, is introduced. It is convenient to let $Rvar \subseteq Uvar$ denote a set of renaming variables that cannot occur in programs, that is $Pvar \cap Rvar = \emptyset$, and suppose that $\Upsilon : Pvar \rightarrow Rvar$.

Definition 2 $unify$**.** The partial mapping $unify : Atom \times Subst/\!\approx \times Atom \times Subst/\!\approx \rightarrow Subst/\!\approx$ is defined by:

$$unify(a, [\phi]_{\approx}, b, [\psi]_{\approx}) = [(\varphi \circ \phi) \upharpoonright Pvar]_{\approx} \text{ where } \varphi \in mgu(\{\phi(a) = \Upsilon(\psi(b))\})$$

To approximate the $unify$ operation it is convenient to introduce a collecting semantics, concerned with sets of substitutions, to record the substitutions that occur at various program points. In the collecting semantics interpretation, $unify$ is extended to $unify^c$, which manipulates (possibly infinite) sets of substitutions.

Definition 3 $unify^c$**.** The mapping $unify^c : Atom \times \wp(Subst/\!\approx) \times Atom \times \wp(Subst/\!\approx) \rightarrow \wp(Subst/\!\approx)$ is defined by:

$$unify^c(a, \Phi, b, \Psi) = \{[\theta]_{\approx} \mid [\phi]_{\approx} \in \Phi \wedge [\psi]_{\approx} \in \Psi \wedge [\theta]_{\approx} = unify(a, [\phi]_{\approx}, b, [\psi]_{\approx})\}$$

2.3 Linearity and substitutions

To be more precise about linearity, it is necessary to introduce the variable multiplicity of a term t, denoted $\chi(t)$.

Definition 4 variable multiplicity, χ [5]. The variable multiplicity operator $\chi : Term \rightarrow \{0, 1, 2\}$ is defined by:

$$\chi(t) = \max(\{\chi_u(t) \mid u \in Uvar\}) \text{ where } \chi_u(t) = \begin{cases} 0 \text{ if } u \text{ does not occur in } t \\ 1 \text{ if } u \text{ occurs only once in } t \\ 2 \text{ if } u \text{ occurs many times in } t \end{cases}$$

If $\chi(t) = 0$, t is ground; if $\chi(t) = 1$, t is linear; and if $\chi(t) = 2$, t is non-linear. The significance of linearity is that the unification of linear terms only yields restricted forms of aliasing. Lemma 5 states some of the restrictions on a most general unifier which follow from unification with a linear term.

Lemma 5. $\chi(b) \neq 2 \wedge var(a) \cap var(b) = \emptyset \wedge \phi \in mgu(\{a = b\}) \Rightarrow$

1. $\forall u \in Uvar \,.\, \chi(\phi(u)) = 2 \Rightarrow u \in var(b)$
2. $\forall u, u' \in Uvar \,.\, u \neq u' \wedge var(\phi(u)) \cap var(\phi(u')) \neq \emptyset \Rightarrow u \notin var(a) \vee u' \notin var(a)$.
3. $\forall u', u'' \in var(b) \,.\, u' \neq u'' \wedge w \in var(\phi(u')) \cap var(\phi(u'')) \Rightarrow \exists u \in var(a) .\, \chi_u(a) = 2 \wedge w \in var(\phi(u))$

Application of lemma 5 is illustrated in example 1.

Example 1. Note that $\phi \in mgu(\{f(u,v,v) = f(x,y,z)\})$ where $\phi = \{v \mapsto y, x \mapsto u, z \mapsto y\}$, $\chi(f(x,y,z)) \neq 2$ and that $f(u,v,v)$ and $f(x,y,z)$ do not share variables. Observe that

1. The variables u and v of $f(u,v,v)$ remain linear after unification, that is, $\chi(\phi(u)) = 1$ and $\chi(\phi(v)) = 1$, as predicted by case 1 of lemma 5.
2. The variables of $f(u,v,v)$, specifically u and v, remain unaliased after unification. Indeed, case 2 of lemma 5 asserts that since $u, v \in var(f(u,v,v))$, $var(\phi(u)) \cap var(\phi(v)) = \emptyset$.
3. Informally, case 3 of lemma 5 states that the aliasing which occurs between the variables of $f(x,y,z)$, is induced by a variable of $f(u,v,v)$ which has a multiplicity of 2. For instance, $y \in var(\phi(y)) \cap var(\phi(z))$ with $\chi_v(f(u,v,v)) = 2$ and $y \in var(\phi(v))$.

Lemma 5 differs from the corresponding lemma in [5] (lemma 2.2) in two ways. First, lemma 5 requires that a and b do not share variables. This is essentially a work-around for a subtle mistake in lemma 2.2 [9]. Second, lemma 5 additionally states that a variable which only occurs once in a can only be aliased to one variable in b. This observation permits linearity to be exploited further than in the original proposals for tracking sharing with linearity [22, 5] by putting a tighter constraint of the form of aliasing that occurs on unification with a linear term. The proof for lemma 5 follows by induction on the steps of the unification algorithm.

3 Abstracting substitutions

Sharing analysis is primarily concerned with characterising the sharing effects that can arise among program variables. Correspondingly, abstract substitutions are formulated in terms of sharing groups [14] which represent which program variables share variables. Formally, an abstract substitution is structured as a set of sharing groups where a sharing group is a (possibly empty) set of program variable and linearity pairs.

Definition 6 Occ_{Svar}**.** The set of sharing groups, $(o \in) Occ_{Svar}$ is defined by:

$$Occ_{Svar} = \{o \in \wp(Svar \times \{1,2\}) \mid \forall u \in Svar \,.\, \langle u, 1\rangle \notin o \vee \langle u, 2\rangle \notin o\}$$

$Svar$ is a finite set of program variables. The intuition is that a sharing group records which program variables are bound to terms that share a variable. Additionally, a sharing group expresses how many times the shared variable occurs in the terms to which the program variables are bound. Specifically, a program variable is paired with 1 if it is bound to a term in which the shared variable only occurs once. The variable is paired with 2 if it can be bound to a term in which the shared variable occurs possibly many times. The finiteness of Occ_{Svar} follows from the finiteness of $Svar$. ($Svar$ usually corresponds to $Pvar$, the set of program variables. It is necessary to parameterise Occ, however, so that abstract substitutions are well-defined under renaming by Υ. Then $Svar = Rvar$.)

The precise notion of abstraction is first defined for a single substitution via lin and then, by lifting lin, generalised to sets of substitutions.

Definition 7 *occ* **and** *lin***.** The abstraction mappings $occ : Uvar \times Subst \rightarrow Occ_{Svar}$ and $lin : Subst/\!\approx\, \rightarrow \wp(Occ_{Svar})$ are defined by:

$$occ(u, \phi) = \{\langle v, \chi_u(\phi(v))\rangle \mid u \in var(\phi(v)) \wedge v \in Svar\}$$

$$lin([\phi]_{\approx}) = \{occ(u, \phi) \mid u \in Uvar\}$$

The mapping lin is well-defined since $lin([\phi]_{\approx}) = lin([\varphi]_{\approx})$ if $\phi \approx \varphi$. The mapping occ is defined in terms of $Svar$ because, for the purposes of analysis, the only significant bindings are those which relate to the program variables (and renamed program variables). Note that $\emptyset \in lin([\phi]_{\approx})$ since the codomain of a substitution is always finite.

The abstraction lin is analogous to the abstraction $\mathcal{A}$ used in [21] and implicit in [14]. Both abstractions are formulated in terms of sharing groups. The crucial difference is that lin, as well as expressing sharing, additionally represents linearity information.

Example 2. Suppose $Svar = \{u, v, w, x, y, z\}$ and $\phi = \{u \mapsto u_1,\ w \mapsto v,\ x \mapsto f,\ y \mapsto g(u_1, u_2, u_2),\ z \mapsto h(u_2, u_3, u_3)\}$ then

$$\begin{gathered} lin([\phi]_{\approx}) = \{\emptyset, occ(u_1, \phi), occ(u_2, \phi), occ(u_3, \phi), occ(v, \phi)\} = \\ \{\emptyset, \{\langle u, 1\rangle, \langle y, 1\rangle\}, \{\langle y, 2\rangle, \langle z, 1\rangle\}, \{\langle z, 2\rangle\}, \{\langle v, 1\rangle, \langle w, 1\rangle\}\} \end{gathered}$$

since $occ(w, \phi) = occ(x, \phi) = occ(y, \phi) = occ(z, \phi) = \emptyset$. The salient properties of ϕ, namely sharing, groundness and linearity, are all captured by $lin([\phi]_{\approx})$. The variables of $Svar$ which ϕ ground, do not appear in $lin([\phi]_{\approx})$; and the variables of $Svar$ which are independent (unaliased), never occur in the same sharing group of $lin([\phi]_{\approx})$. Thus $lin([\phi]_{\approx})$ indicates that x is ground and that, for example, v and y are independent. Additionally, $lin([\phi]_{\approx})$ captures the fact that grounding either v or w grounds the other. Or, put another way, that v and w are strongly coupled [25].

Linearity is also represented and $lin([\phi]_{\approx})$ indicates that $\chi(\phi(x)) = 0$; $\chi(\phi(u)) = \chi(\phi(v)) = \chi(\phi(w)) = 1$; and $\chi(\phi(y)) = \chi(\phi(z)) = 2$. It is evident that $\chi(\phi(w)) = 1$, for instance, since $\chi_v(\phi(w)) = 1$ and $\chi_u(\phi(w)) \neq 2$ for all $u \in Uvar$. Specifically, $\langle w, 1\rangle \in occ(v, \phi)$ and $\langle w, 2\rangle \notin occ(u, \phi)$ for all $u \in Uvar$. The subtlety is that the domain represents variable multiplicity information slightly more accurately than the Søndergaard domain [22, 5]. Note that although $\chi(\phi(y)) = 2$ and y is aliased to both u and z, $lin([\phi]_{\approx})$ indicates that the variable that occurs through u and y (namely u_1) occurs only once in $\phi(y)$ whereas the variable through y and z (that is to say u_2) occurs multiply in $\phi(y)$. This can be exploited to gain more precise analysis.

The abstract domain, the set of abstract substitutions, is defined below using the convention that abstractions of concrete objects and operations are distinguished with a $*$ from the corresponding concrete object or operation.

Definition 8 $Subst^{*}_{Svar}$**.** The set of abstract substitutions, $Subst^{*}_{Svar}$, is defined by: $Subst^{*}_{Svar} = \wp(Occ_{Svar})$.

Like previous sharing groups domains [14, 21], $Subst^{*}_{Svar}(\subseteq)$ is a finite lattice with set union as the lub. $Subst^{*}_{Svar}$ is finite since Occ_{Svar} is finite.

The *lin* abstraction naturally lifts to sets of substitutions, but to define concretisation, the notion of approximation implicit in linearity (specifically in the denotations 1 and 2) must be formalised. In the abstraction, a program variable is paired with 1 if it is definitely bound to a term in which the shared variable only occurs once; and is paired with 2 if it can possibly be bound to a term in which the shared variable occurs multiply. This induces the poset $Occ_{Svar}(\leq)$ defined by: $o \leq o'$ if and only if $var(o) = var(o')$ and for all $\langle u, m \rangle \in o$ there exists $\langle u, m' \rangle \in o'$ such that $m \leq m'$. The poset lifts to the preorder $Subst^{*}_{Svar}(\leq)$ by: $\phi^{*} \leq \phi'^{*}$ if and only if for all $o \in \phi^{*}$ there exists $o' \in \phi'^{*}$ such that $o \leq o'$.

Definition 9 α_{lin} **and** γ_{lin}**.** The abstraction and concretisation mappings $\alpha_{lin} : \wp(Subst/\approx) \to Subst^{*}_{Svar}$ and $\gamma_{lin} : Subst^{*}_{Svar} \to \wp(Subst/\approx)$ are defined by:

$$\alpha_{lin}(\Phi) = \cup_{[\phi]_{\approx} \in \Phi} lin([\phi]_{\approx}), \quad \gamma_{lin}(\phi^{*}) = \{[\phi]_{\approx} \in Subst/\approx \mid lin([\phi]_{\approx}) \leq \phi^{*}\}$$

The structure of α_{lin} and γ_{lin} mirrors that of the abstraction and concretisation operations found in [14, 21].

As illustrated in example 2, the *lin* abstraction can encode the variable multiplicity of a substitution. More significantly, if $\phi \in \gamma_{lin}(\phi^{*})$, the variable multiplicity of $\phi(t)$ can be (partially) deduced from t and ϕ^{*}. The precise relationship between $\chi(\phi(t))$ and t and ϕ^{*} is formalised in definition 10 and lemma 11, with an analog of χ, denoted χ^{*}.

Definition 10 χ^{*}**.** The abstract variable multiplicity operator $\chi^{*} : Term \times Occ_{Svar} \to \{0, 1, 2\}$ is defined by:

$$\chi^{*}(t, o) = \begin{cases} 0 & \text{if } \forall\, v \in var(o) \,.\, \chi_v(t) = 0 \\ 2 & \text{if } \exists\, v \in var(o) \,.\, \chi_v(t) = 2 \\ 2 & \text{if } \exists\, v, v' \in var(t) \,.\, v, v' \in var(o) \wedge v \neq v' \\ 2 & \text{if } \exists\, v \in var(t) \,.\, \langle v, 2 \rangle \in o \\ 1 & \text{otherwise} \end{cases}$$

Lemma 11.

$$var(t) \subseteq Svar \wedge occ(u, \phi) \leq o \Rightarrow \chi_u(\phi(t)) \leq \chi^{*}(t, o)$$

To conservatively calculate the variable multiplicity of a term t in the context of a set of substitutions represented by ϕ^{*}, the sharing group operator χ^{*} is lifted to abstract substitutions via *ln* and *nl*.

Definition 12 *ln* **and** *nl***.** The mappings $ln : Term \times Subst^*_{Svar} \to Subst^*_{Svar}$ and $nl : Term \times Subst^*_{Svar} \to Subst^*_{Svar}$ are defined by:

$$ln(t, \phi^*) = \{o \in \phi^* \mid \chi^*(t, o) = 1\}, \quad nl(t, \phi^*) = \{o \in \phi^* \mid \chi^*(t, o) = 2\}$$

The operators ln and nl essentially categorise ϕ^* into two sorts of sharing group: sharing groups which describe aliasing for which $\phi(t)$ is definitely linear; and sharing groups which represent aliasing for which $\phi(t)$ is possibly non-linear. An immediate corollary of lemma 11, corollary 13, asserts that $\phi(t)$ is linear if $nl(t, \phi^*)$ is empty.

Corollary 13.

$$[\phi]_{\approx} \in \gamma_{lin}(\phi^*) \wedge var(t) \subseteq Svar \wedge nl(t, \phi^*) = \emptyset \Rightarrow \chi(\phi(t)) \neq 2$$

The significance of corollary 13 is that it explains how by inspecting t and ϕ^*, $\phi(t)$ can be inferred to be linear, thereby enabling linear instances of unification to be recognised.

4 Abstracting unification

The collecting version of the $unify$ operator, $unify^c$, provides a basis for abstracting the basic operations of logic programming by spelling out how to manipulate (possibly infinite) sets of substitutions. The usefulness of the collecting semantics as a form of program analysis, however, is negated by the fact that it can lead to non-terminating computations. Therefore, in order to define a practical analyser it is necessary to finitely abstract $unify^c$. To synthesise a sharing analysis, an analog of $unify^c$, $unify^*$, is introduced to manipulate sets of substitutions following the abstraction scheme prescribed by α_{lin} and γ_{lin}.

Just as $unify^c$ is defined in terms of mgu, $unify^*$ is defined in terms of an abstraction of mgu, mge, which traces the steps of the unification algorithm. The unification algorithm takes as input, E, a set of unification equations. E is recursively transformed to a set of simplified equations which assume the form $v = v'$ or $v = \tau_n$. These simplified equations are then solved. The equation solver mge, adopts a similar strategy, but relegates the solution of the simplified equations to $solve$. The skeleton of the abstract equation solver mge is given below in definition 14.

Definition 14 *mge***.** The relation $mge : Eqn \times Subst^*_{Svar} \times Subst^*_{Svar}$ is defined by:

$$\begin{array}{rll}
mge(\emptyset, \sigma^*, \sigma^*) & & \\
mge(v = v' : E, \sigma^*, \theta^*) & \text{if } mge(E, \sigma^*, \theta^*) \wedge & v \equiv v' \\
mge(v = v' : E, \sigma^*, \theta^*) & \text{if } mge(E, solve(v, v', \sigma^*), \theta^*) \wedge & v \not\equiv v' \\
mge(v = \tau_n : E, \sigma^*, \theta^*) & \text{if } mge(E, solve(v, \tau_n, \sigma^*), \theta^*) \wedge & v \notin var(\tau_n) \\
mge(\tau_n = v : E, \sigma^*, \theta^*) & \text{if } mge(v = \tau_n : E, \sigma^*, \theta^*) & \\
mge(\tau_n = \tau_n^* : E, \sigma^*, \theta^*) & \text{if } mge(t_1 = t_1' : \ldots : t_n = t_n' : E, \sigma^*, \theta^*) \wedge f \equiv f' &
\end{array}$$

To spare the need to define an extra (composition) operator for abstract substitutions, mge is defined to abstract a variant of mgu. Specifically, if $\varphi \in mgu(\{\phi(a) = \phi(b)\})$, $[\phi]_{\approx} \in \gamma_{lin}(\phi^{\star})$, and $mge(\{a = b\}, \phi^{\star}, \mu^{\star})$, then $\mu^{\star}$ abstracts the composition $\varphi \circ \phi$ (rather than φ), that is, $[\varphi \circ \phi]_{\approx} \in \gamma_{lin}(\mu^{\star})$.

To define $solve$, and thereby mge, a number of auxiliary operators are required. The first, denoted $rl(t, \phi^{\star})$, represents the sharing groups of $\phi^{\star}$ which are relevant to the term t, that is, those sharing groups of $\phi^{\star}$ which share variables with t.

Definition 15 *rl* **[14].** The mapping $rl : Term \times Subst^{\star}_{Svar} \rightarrow Subst^{\star}_{Svar}$ is defined by: $rl(t, \phi^{\star}) = \{o \in \phi^{\star} \mid var(o) \cap var(t) \neq \emptyset\}$.

Note that $rl(t, \phi^{\star}) = \{o \in \phi^{\star} \mid \chi^{\star}(t, o) \neq 0\}$ and therefore $rl(t, \phi^{\star}) = ln(t, \phi^{\star}) \cup nl(t, \phi^{\star})$. In [14] the equivalent operator is denoted rel.

The second operator, $\sqcup$, is a technical device which is used to calculate $occ(u, \varphi \circ \phi)$ from a set of sharing groups $occ(w, \phi)$ for the variables w with $u \in var(\varphi(w))$. Since $occ(u, \varphi \circ \phi) = \{\langle v, \chi_u(\varphi \circ \phi(v))\rangle \mid u \in var(\varphi \circ \phi(v)) \wedge v \in Svar\}$, observe that $\langle v, 1\rangle \in occ(u, \varphi \circ \phi)$ if a single variable w satisfies $u \in var(\varphi(w))$ and additionally $\chi_w(\phi(v)) = 1$ with $\chi_u(\varphi(w)) = 1$. Otherwise $\langle v, 2\rangle \in occ(u, \varphi \circ \phi)$ if there exist distinct variables w and w' for which $u \in var(\varphi(w)) \cap var(\varphi(w'))$, or $\chi_w(\phi(v)) = 2$, or $\chi_u(\varphi(w)) = 2$. Thus $\langle v, \min(\Sigma_{u \in var(\varphi(w))} m_{v,w}, 2)\rangle \in occ(u, \varphi \circ \phi)$ where $m_{v,w} = \max(\chi_u(\varphi(w)), \chi_w(\phi(v)))$. The rôle of the $\sqcup$ operator is to compute $occ(u, \varphi \circ \phi)$ by calculating the pairs $\langle v, \min(\Sigma_{u \in var(\varphi(w))} m_{v,w},\ 2)\rangle$ given $m_{v,w}$ for $u \in var(\varphi(w))$.

Definition 16 $\sqcup$**.** The operator $\sqcup \cdot : \wp(Occ_{Svar}) \rightarrow Occ_{Svar}$ is defined by:

$$\sqcup_{w \in W} o_w = \{\langle v, \min(\Sigma_{\langle v, m_{v,w}\rangle \in o_w} m_{v,w}, 2)\rangle \mid v \in \cup_{w \in W} var(o_w)\}$$

Although the motivation for $\sqcup$ is technical, example 3 illustrates that the operator itself is straightforward to use and compute. Sometimes, for brevity, $\sqcup$ is written infix.

Example 3. Three examples of using the $\sqcup$ operator are given below: first, $\{\langle u, 1\rangle, \langle v, 1\rangle, \langle w, 2\rangle\} \sqcup \{\langle v, 1\rangle, \langle w, 2\rangle, \langle x, 2\rangle, \langle y, 1\rangle\} = \{\langle u, \min(1, 2)\rangle, \langle v, \min(1+1, 2)\rangle, \langle w, \min(2+2, 2)\rangle, \langle x, \min(2, 2)\rangle, \langle y, \min(1, 2)\rangle\} = \{\langle u, 1\rangle, \langle v, 2\rangle, \langle w, 2\rangle, \langle x, 2\rangle, \langle y, 1\rangle\}$; second, $\emptyset \sqcup \emptyset = \emptyset$; and third, $\sqcup_{w \in \emptyset} o_w = \emptyset$.

Note that $\sqcup$ is commutative and associative but is not idempotent, and specifically, $o \sqcup o = var(o) \times \{2\}$. Also observe that $var(\sqcup_{w \in W} o_w) = \cup_{w \in W} var(o_w)$ hinting at the fact that $\sqcup$ generalises set union which is used to combine sharing groups in the original sharing analyses [14, 21].

In the conventional approach, worst-case aliasing is always assumed and a closure under union operator is used to enumerate all the possible sharing groups that can possibly arise in unification [14, 21]. The $\sqcup$ operator defines an analog of closure under union, closure under $\sqcup$, denoted $\phi^{\star\star}$ and defined in definition 17.

Definition 17 closure under $\sqcup$, *. The closure under $\sqcup$ operator $\cdot^\star : Subst^*_{Svar} \to Subst^*_{Svar}$ is defined by: $\phi^{*\star} = \phi^* \cup \{o \sqcup o' \mid o, o' \in \phi^{*\star}\}$.

Closure under $\sqcup$ is used more conservatively than the closure under union operator of [14, 21] and is only invoked in the absence of useful linearity information. An interesting consequence of $Subst^*_{Svar}(\leq)$ being a preorder (rather than a poset), is that equivalent $\phi^{*\star}$ can have different representations. For instance, if $\phi^* = \{\{\langle u,1\rangle, \langle v,2\rangle\}\}$, $\phi^{*\star} = \{\{\langle u,1\rangle, \langle v,2\rangle\}, \{\langle u,2\rangle, \langle v,2\rangle\}\}$ but $\varphi^{*\star} \leq \phi^{*\star} \leq \varphi^{*\star}$ where $\varphi^* = \{\{\langle u,2\rangle, \langle v,2\rangle\}\}$ and $\varphi^{*\star} = \{\{\langle u,2\rangle, \langle v,2\rangle\}\}$. Clearly $\varphi^{*\star}$ is preferable to $\phi^{*\star}$, and more generally, redundancy can be avoided in the calculation and representation of $\phi^{*\star}$ by computing $\phi^{*\star}$ with $\{var(o) \times \{2\} \mid o \in \phi^*\}^\star$.

Finally, to achieve a succinct definition of the abstract equation solver, it is useful to lift $\sqcup$ to sets of sharing groups in the matter prescribed in definition 18.

Definition 18 $\Box$. The mapping $\cdot \Box \cdot : Subst^*_{Svar} \times Subst^*_{Svar} \to Subst^*_{Svar}$ is defined by: $\phi^* \Box \phi'^* = \{o \sqcup o' \mid o \in \phi^* \wedge o' \in \phi'^*\}$.

The nub of the equation solver mge is $solve$. In essence, $solve(v, t, \phi^*)$ solves the syntactic equation $v = t$ in the presence of the abstract substitution ϕ^*, returning the composition of the unifier with ϕ^*. The different cases of operator $solve$ apply different analysis strategies corresponding to when $\phi(v)$ is linear, $\phi(t)$ is linear, both $\phi(v)$ and $\phi(t)$ are possibly non-linear. (If both $\phi(v)$ and $\phi(t)$ are linear, cases 1 and 2 coincide.) The default strategy corresponds to the standard treatment of the abstract solver $amgu$ of [14].

Definition 19 $solve$. The abstract equation solver $solve : Uvar \times Term \times Subst^*_{Svar} \to Subst^*_{Svar}$ is defined by:

$$solve(v, t, \phi^*) = \phi^* \setminus (rl(v, \phi^*) \cup rl(t, \phi^*)) \cup$$
$$\begin{cases} (ln(v, \phi^*) \Box ln(t, \phi^*)) \cup (ln(v, \phi^*)^\star \Box nl(t, \phi^*)) & \text{if } nl(v, \phi^*) = \emptyset \wedge \\ & \quad ln(v, \phi^*) \cap rl(t, \phi^*) = \emptyset \\ (ln(v, \phi^*) \Box ln(t, \phi^*)) \cup (nl(v, \phi^*) \Box ln(t, \phi^*)^\star) & \text{if } nl(t, \phi^*) = \emptyset \wedge \\ & \quad ln(t, \phi^*) \cap rl(v, \phi^*) = \emptyset \\ rl(v, \phi^*)^\star \ \Box \ rl(t, \phi^*)^\star & \text{otherwise} \end{cases}$$

Note that $\phi^* \Box \emptyset = \emptyset$ and $\emptyset \Box \phi^* = \emptyset$ and in particular, for case 1 of $solve$, the closure $ln(v, \phi^*)^\star$ need not be calculated if $nl(t, \phi^*) = \emptyset$. Similarly, in case 2, if $nl(v, \phi^*) = \emptyset$, $ln(t, \phi^*)^\star$ need not be computed. The correctness of $solve$ is asserted by lemma 20. The justification of lemma 20 relies on very weak properties of substitutions, and specifically, only that a most general unifier, if it exists, is idempotent.

Lemma 20.

$$\begin{array}{l}[\phi]_\approx \in \gamma_{lin}(\phi^*) \wedge \varphi \in mgu(\{\phi(v) = \phi(t)\}) \wedge \\ \quad \{v\} \cup var(t) \subseteq Svar \wedge v \notin var(t) \Rightarrow [\varphi \circ \phi]_\approx \in \gamma_{lin}(solve(v, t, \phi^*))\end{array}$$

The correctness of mge follows from lemma 20 and is stated as corollary 21.

Corollary 21.

$$[\phi]_{\approx} \in \gamma_{lin}(\phi^\star) \wedge \varphi \in mgu(\phi(E)) \wedge$$
$$mge(E, \phi^\star, \mu^\star) \wedge var(E) \subseteq Svar \Rightarrow [\varphi \circ \phi]_{\approx} \in \gamma_{lin}(\mu^\star)$$

It is convenient to regard mge as a mapping, that is, $mge(E, \phi^\star) = \mu^\star$ if $mge(E, \phi^\star, \mu^\star)$. Strictly, it is necessary to show that $mge(E, \phi^\star, \mu^\star)$ is deterministic for $mge(E, \phi^\star)$ to be well-defined. Like in [5], the conjecture is that mge yields a unique abstract substitution regardless of the order in which E is solved. This conjecture, however, is only really of theoretical interest because all that really matters is that any abstract substitution derived by mge is safe. This is essentially what corollary 21 asserts.

To define $unify^\star$, the finite analog of $unify^c$, it is necessary to introduce an abstract restriction operator, denoted $\cdot \restriction^\star \cdot$.

Definition 22 abstract restriction, $\restriction^\star$. The abstract restriction operator $\cdot \restriction^\star \cdot : Subst^\star_{Svar} \times \wp(Uvar) \rightarrow Subst^\star_{Svar}$ is defined by: $\phi^\star \restriction^\star U = \{o \restriction^\star U \mid o \in \phi^\star\}$ where $o \restriction^\star U = \{\langle u, m\rangle \in o \mid u \in U\}$.

The definition of $unify^\star$ is finally given below, followed by the local safety theorem, theorem 24.

Definition 23 $unify^\star$. The mapping $unify^\star : Atom \times Subst^\star_{Pvar} \times Atom \times Subst^\star_{Pvar} \rightarrow Subst^\star_{Pvar}$ is defined by:

$$unify^\star(a, \phi^\star, b, \psi^\star) = mge(\{a = \Upsilon(b)\}, \phi^\star \cup \Upsilon(\psi^\star)) \restriction^\star Pvar$$

Theorem 24 local safety of $unify^\star$.

$$\Phi \subseteq \gamma_{lin}(\phi^\star) \wedge \Psi \subseteq \gamma_{lin}(\psi^\star) \wedge$$
$$var(a) \cup var(b) \subseteq Pvar \Rightarrow unify^c(a, \Phi, b, \Psi) \subseteq \gamma_{lin}(unify^\star(a, \phi^\star, b, \psi^\star))$$

Examples 4 and 5 demonstrate the precision in propagating groundness information that the domain inherits from sharing groups, and accuracy that is additionally obtained by tracking linearity. Furthermore, example 6 illustrates that the domain is more powerful than the sum of its parts, that is, it can trace linearity and sharing better than is achievable by running the Søndergaard [22, 5] and sharing group analyses [14, 21] together in lock step [6]. The examples also comment on the efficiency of the analysis.

Example 4 propagating groundness. The supremacy of the sharing group domains over the Søndergaard domain for propagating groundness information can be illustrated by separately solving two equations, first, $x = f(y, z)$ and second, $x = f(g, g)$. Suppose $Svar = \{x, y, z\}$. To demonstrate the groundness propagation of sharing groups, let $\phi^\star = \{\emptyset, \{\langle x, 2\rangle\}, \{\langle y, 2\rangle\}, \{\langle z, 2\rangle\}\}$ so that worst-case linearity is assumed. Solving $x = f(y, z)$ for $\phi^\star$ yields

$$\varphi^\star = solve(x, f(y, z), \phi^\star) =$$
$$\{\emptyset, \{\langle x, 2\rangle, \langle y, 2\rangle\}, \{\langle x, 2\rangle, \langle z, 2\rangle\}, \{\langle x, 2\rangle, \langle y, 2\rangle, \langle z, 2\rangle\}\}$$

Since x occurs in each (non-empty) sharing group of φ^*, grounding x must also ground both y and z, and indeed $\psi^* = solve(x, f(g, g), \phi^*) = \{\emptyset\}$. Furthermore, ψ^* indicates that y and z are independent. In contrast, the abstract unification algorithm proposed for the Søndergaard domain [5], cannot infer that x and y are grounded or independent.

Example 5 tracking linearity. Suppose $E = \{x = u,\ y = f(u, v),\ z = v\}$ and consider the abstraction of $mgu(E)$ and specifically the calculation $mge(E, lin([\epsilon]_\approx))$. Assuming $Svar = \{u,\ v,\ x,\ y,\ z\}$, dubbing $\epsilon^* = lin([\epsilon]_\approx) = \{\emptyset,\ \{\langle u, 1\rangle\},\ \{\langle v, 1\rangle\},\ \{\langle x, 1\rangle\},\ \{\langle y, 1\rangle\},\ \{\langle z, 1\rangle\}\}$, and solving the equations left-to-right

$$\begin{aligned}
\phi^* &= solve(x, u, \epsilon^*) &&= \{\emptyset, \{\langle u, 1\rangle, \langle x, 1\rangle\}, \{\langle v, 1\rangle\}, \{\langle y, 1\rangle\}, \{\langle z, 1\rangle\}\} \\
\varphi^* &= solve(y, f(u, v), \phi^*) &&= \{\emptyset, \{\langle u, 1\rangle, \langle x, 1\rangle, \langle y, 1\rangle\}, \{\langle v, 1\rangle, \langle y, 1\rangle\}, \{\langle z, 1\rangle\}\} \\
\psi^* &= solve(z, v, \varphi^*) &&= \{\emptyset, \{\langle u, 1\rangle, \langle x, 1\rangle, \langle y, 1\rangle\}, \{\langle v, 1\rangle, \langle y, 1\rangle, \langle z, 1\rangle\}\}
\end{aligned}$$

Therefore $\psi^* = mge(E, \epsilon^*)$ and indeed $\psi = \{x \mapsto u, y \mapsto f(u, v), z \mapsto v\} \in mgu(E)$ with $[\psi]_\approx \in \gamma_{lin}(\psi^*)$. Without exploiting linearity (or freeness), the sharing group analyses of [14, 21] have to include an additional sharing group $\{u,\ v,\ x,\ y,\ z\}$ for possible aliasing between u and v (and x and z). Tracking linearity strengthens the analysis, allowing it to deduce that u and v (and x and z) are definitely not aliased. Note also that the size of the data structure (the abstract substitution ψ^*) is pruned from 4 to 3 sharing groups and that, in contrast to the analyses of [14, 21], the calculation of a closure is avoided.

Example 6 refined sharing and linearity. The domain refines the way linearity information is recorded and in particular the analysis can differentiate between which variables can occur multiply in a term (or binding) and which variables always occur singly in a term (or binding). For instance, consider the set of substitutions $\Phi = \{[\phi]_\approx, [\phi']_\approx\}$ where $\phi = \{x \mapsto f(u, v)\}$ and $\phi' = \{x \mapsto f(w, w)\}$. Φ represents two possible bindings for x. In the first, $\phi(x)$ is linear, whereas in the second, $\phi'(x)$ is non-linear. This is reflected in $\phi^* = \alpha_{lin}(\Phi) = lin([\phi]_\approx) \cup lin([\phi']_\approx)$, and specifically, if $Svar = \{u,\ v,\ w,\ x,\ y,\ z\}$

$$\phi^* = \{\emptyset, \{\langle u, 1\rangle, \langle x, 1\rangle\}, \{\langle v, 1\rangle, \langle x, 1\rangle\}, \{\langle w, 1\rangle, \langle x, 2\rangle\}, \{\langle y, 1\rangle\}, \{\langle z, 1\rangle\}\}$$

The abstraction ϕ^* indicates that u and v never occur more than once through $\phi(x)$ and $\phi'(x)$, and that w can occur multiply through $\phi(x)$ or $\phi'(x)$. Informally, the abstraction records why x is possibly non-linear. This, in turn, can lead to improved precision and efficiency, as is illustrated by the calculation of $mge(\{x = f(y, z), w = g\}, \phi^*)$. Again, solving the equations left-to-right

$$\begin{aligned}
\varphi^* = solve(x, f(y, z), \phi^*) &= \{\emptyset, \{\langle u, 1\rangle, \langle x, 1\rangle, \langle y, 1\rangle\}, \{\langle u, 1\rangle, \langle x, 1\rangle, \langle z, 1\rangle\}, \\
&\qquad \{\langle v, 1\rangle, \langle x, 1\rangle, \langle y, 1\rangle\}, \{\langle v, 1\rangle, \langle x, 1\rangle, \langle z, 1\rangle\}, \\
&\qquad \{\langle w, 1\rangle, \langle x, 2\rangle, \langle y, 2\rangle\}, \{\langle w, 1\rangle, \langle x, 2\rangle, \langle z, 2\rangle\}, \\
&\qquad \{\langle w, 1\rangle, \langle x, 2\rangle, \langle y, 2\rangle, \langle z, 2\rangle\}\} \\
\psi^* = solve(w, g, \varphi^*) &= \{\emptyset, \{\langle u, 1\rangle, \langle x, 1\rangle, \langle y, 1\rangle\}, \{\langle u, 1\rangle, \langle x, 1\rangle, \langle z, 1\rangle\}, \\
&\qquad \{\langle v, 1\rangle, \langle x, 1\rangle, \langle y, 1\rangle\}, \{\langle v, 1\rangle, \langle x, 1\rangle, \langle z, 1\rangle\}\}
\end{aligned}$$

In terms of precision, linearity is still exploited for u and v, even though worst-case aliasing has to be assumed for w. Consequently, on grounding w, u and v (and y and z) become independent. The Søndergaard domain, however, cannot resolve linearity to the same degree of accuracy and therefore the analysis of [5] cannot infer u and v (and y and z) become unaliased. Also, the combined domains approach [6] does not help, since the precision comes from restructuring the domain. In terms of efficiency, observe that although the closure of $ln(f(y, z), \phi^*)$ is computed, the number of sharing groups in $\varphi^\star$ is kept low by only combining $ln(f(y, z), \phi^*)^\star$ with $nl(x, \phi^*)$ (rather than with $rl(x, \phi^*)$).

The extra expressiveness of the domain is not confined to abstracting multiple substitutions. If $\mu = \{x \mapsto f(u, v, w, w)\}$ and $\mu^* = lin([\mu]_{\approx})$, for instance,

$$\mu^* = \{\emptyset, \{\langle u, 1\rangle, \langle x, 1\rangle\}, \{\langle v, 1\rangle, \langle x, 1\rangle\}, \{\langle w, 1\rangle, \langle x, 2\rangle\}, \{\langle y, 1\rangle\}, \{\langle z, 1\rangle\}\}$$

so that μ^* is structurally identical to ϕ^*. Although omitted for brevity, the calculation $mge(\{x = f(y_1, y_1, y_3, y_4), w = g\}, \mu^*)$ deduces that y_i and y_j (for $i \neq j$) become independent after w is grounded. This, again, cannot be inferred in terms of the Søndergaard domain.

5 Related work

Recently, four interesting proposals for computing accurate sharing information have been put forward in the literature. In the first proposal [6], domains and analyses are combined to improve accuracy. This paper develops this theme and explores the virtues of fusing linearity with sharing groups. In short, this paper explains how accuracy and efficiency can be further improved by restructuring a combined domain as a single domain.

In the second proposal [4], the correctness of freeness analyses is considered. An abstract unification algorithm is proposed as a basis for constructing accurate freeness analyses with a domain formulated in terms of abstract equations. Safety follows because the abstract algorithm mimics the solved form algorithm in an intuitive way. Correctness is established likewise here. The essential distinction between the two works is that this paper tracks groundness and linearity. Consequently, the approach presented here can derive more accurate sharing information. Also, as pointed out in [2], "it is doubtful whether it (the abstract unification algorithm of [4]) can be the basis for a very efficient analysis". The analysis presented here, on the other hand, is designed to be efficient.

Very recently, in the third proposal [2], an analysis for sharing, groundness, linearity and freeness is formalised as a transition system which reduces a set of abstract equations to an abstract solved form. Sharing is represented in a sharing group fashion with variables enriched with linearity and freeness information by an annotation mapping. The domain, however, essentially adopts the Jacobs and Langen [14] structure. Consequently the analysis cannot always derive sharing as accurately as the analysis reported here. Moreover, the use of a tightly-coupled domain seems to simplify some of the analysis machinery. For instance, the notion of abstraction introduced in this paper is more succinct than the equivalent definition in [2]. This simplicity seems to stem from the fact the domain is an elegant and natural generalisation of sharing groups [14]. Also, the analysis of [2] has not, as yet, been proved correct.

Fourthly, a referee pointed out a freeness analysis which also tracks linearity to avoid the calculation of closures in sharing groups [11]. Interestingly, [11] seems to adopt a conventional notion of linearity, rather than embedding linearity into sharing groups in the useful way that is described in this paper.

To be fair, however, the analyses of [11, 4, 2] do infer freeness. This can be useful [13]. Although freeness information is not derived in this paper, it seems that freeness can be embedded into sharing groups in a similar way to linearity. What is more, if freeness is recorded this way, it can be used to improve sharing beyond what is achievable by just tracing linearity! This is unusual, contrasts to [2], and is further evidence for the usefulness of restructuring sharing groups.

6 Conclusions

A powerful, formally justified and potentially efficient analysis has been presented for inferring definite groundness and possible sharing between the variables of a logic program. The analysis builds on the combined domain approach [6] by elegantly representing linearity information in a sharing group format. By revising sharing groups to capture linearity, a single coherent domain and analysis has been formulated which more precisely captures aliasing behaviour; propagates groundness information with greater accuracy; and in addition, a yields a more refined notion of linearity. In more pragmatic terms, the analysis permits aliasing and groundness to be inferred to a higher degree of accuracy than in previous proposals. The analysis is significant because sharing information underpins many optimisations in logic programming.

Acknowledgements

Thanks are due to Manuel Hermenegildo and Dennis Dams for useful discussions on linearity. This work was supported by ESPRIT project (6707) "ParForce".

References

1. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *J. Logic Programming*, 10:91–124, 1991.
2. M. Bruynooghe and M. Codish. Freeness, sharing, linearity and correctness – all at once. In *WSA'93*, pages 153–164, September 1993.
3. J.-H. Chang and A. M. Despain. Semi-intelligent backtracking of prolog based static data dependency analysis. In *JICSLP'85*. IEEE Computer Society, 1985.
4. M. Codish, D. Dams, G. Filé, and M. Bruynooghe. Freeness analysis for logic programs - and correctness? In *ICLP'93*, pages 116–131. MIT Press, June 1993.

5. M. Codish, D. Dams, and E. Yardeni. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In *ICLP'91*, pages 79–93, Paris, France, 1991. MIT Press.
6. M. Codish, A. Mulkers, M. Bruynooghe, M. J. García de la Banda, and M. Hermenegildo. Improving abstract interpretation by combining domains. In *PEPM'93*. ACM Press, 1993.
7. A. Cortesi and G. Filé. Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis. In *PEPM'91*, pages 52–61. ACM Press, 1991.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM Press, 1977.
9. D. Dams. Personal communication on linearity lemma 2.2. July, 1993.
10. S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM TOPLAS*, 11(3):418–450, July 1989.
11. W. Hans and S. Winkler. Aliasing and groundness analysis of logic programs through abstract interpretation and its safety. Technical Report Nr. 92-27, RWTH Aachen, Lehrstuhl für Informatik II Ahornstraße 55, W-5100 Aachen, 1992.
12. M. Hermenegildo. Personal communication on freeness analysis. May, 1993.
13. M. Hermenegildo and F. Rossi. Non-strict independent and-parallelism. In *ICLP'90*, pages 237–252, Jerusalem, 1990. MIT Press.
14. D. Jacobs and A. Langen. Static Analysis of Logic Programs. *J. Logic Programming*, pages 154–314, 1992.
15. A. King. A new twist to linearity. Technical Report CSTR 93-13, Department of Electronics and Computer Science, Southampton University, Southampton, 1993.
16. J. Lassez, M. J. Maher, and K. Marriott. *Foundations of Deductive Databases and Logic Programming*, chapter Unification Revisited. Morgan Kaufmann, 1987.
17. B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity. In *ICLP'91*, pages 64–78. MIT Press, 1991.
18. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
19. K. Marriott and H. Søndergaard. Analysis of constraint logic programs. In *NACLP'90*, pages 531–547. MIT Press, 1990.
20. K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In *ICLP'91*, pages 49–63, Paris, France, 1991. MIT Press.
21. K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency through abstract interpretation. *J. of Logic Programming*, pages 315–437, 1992.
22. H. Søndergaard. An application of the abstract interpretation of logic programs: occur-check reduction. In *ESOP'86*, pages 327–338. Springer-Verlag, 1986.
23. R. Sundararajan and J. Conery. An abstract interpretation scheme for groundness, freeness, and sharing analysis of logic programs. In *12^{th} FST and TCS Conference*, New Delhi, India, December 1992. Springer-Verlag.
24. A. Taylor. *High Performance Prolog Implementation*. PhD thesis, Basser Department of Computer Science, Sydney, Australia, July 1991.
25. H. Xia. *Analyzing Data Dependencies, Detecting And-Parallelism and Optimizing Backtracking in Prolog Programs*. PhD thesis, University of Berlin, April 1989.

This article was processed using the LaTeX macro package with LLNCS style