Depth-k Sharing and Freeness

Andy King
Computing Laboratory,
University of Kent, Canterbury, CT2 7NF, UK.

Paul Soper
Department of Electronics and Computer Science,
University of Southampton, Southampton, SO17 1BJ, UK.

Abstract

Analyses for variable sharing and freeness are important both in the automatic parallelisation and in the optimisation of sequential logic programs. In this paper, a new analysis is described which can infer sharing and freeness information to an unusually high degree of accuracy. By encoding structural properties of substitutions in a sharing group fashion, a powerful depth-k sharing and freeness analysis is synthesised which exploits the synergy between tracing sharing information and tracking term structure. The analysis propagates groundness with the accuracy of sharing groups and yet can precisely infer sharing and freeness. 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: the sound removal of the occur-check [15]; specialisation of unification [17]; and the detection [12] and efficient exploitation [9, 13] of independent and-parallelism [8].

This paper is concerned with a semantic basis for sharing and freeness analysis, and in particular, the justification of a precise abstract unification algorithm. The abstract unification algorithm finitely traces unification by representing substitutions with sharing and freeness abstractions. The accuracy of the analysis depends, in part, on the substitution properties that the sharing abstractions capture. For instance, a knowledge of freeness can improve sharing (and *vice versa*) [12]. Freeness [3, 12, 16] relates to the structure of a term or binding. Freeness information distinguishes between a free variable, a variable which is definitely not bound to a non-variable term; and a non-free variable, a variable which is possibly bound to a non-variable term. Without exploiting freeness (or linearity [5, 15]), analyses have to assume that aliasing is transitive. Freeness information, in addition, is essential in the detection of non-strict and-parallelism [8].

Conventional sharing and freeness analyses [2, 3, 6, 7, 12, 16] typically adopt a coarse-grained approach to analysis and do not always adequately reason about the fine-grained sharing and freeness interactions between sub-terms. In some circumstances, however, accuracy can pivot on the ability of an analysis to reason about the sharing and freeness of sub-terms. Put another way, for the required precision, it may be necessary to trace sharing and freeness to depth-k [14].

This paper presents a new approach to sharing and freeness analysis that is capable of reasoning about sharing and freeness to depth-k. The analysis explains how structural properties of substitutions can be represented in a sharing group format [9, 13]. In effect, this is a two-fold win: first, groundness and sharing is improved; second, freeness can be refined. The analysis has also been proven correct. This is important because subtle errors and omissions have been reported [6] in some of the more recent proposals for freeness analysis [3, 12, 16]. Thus formal proof is useful, indeed necessary, to instill confidence. The exposition is structured as follows. Section 2 describes the notation and preliminary definitions which will be used throughout. The depth-k analysis is constructed in two parts to ease its development and justification. Section 3 develops a depth-∞ framework for sharing and (possible) freeness analysis. The framework explains how to abstract structural properties of substitutions with sharing groups. The framework, alas, can lead to unterminating computations. Section 4 is thus concerned with finiteness, detailing how to collapse the depth-∞ framework into a tractable and practical depth-k analysis. To trace definite freeness, possible groundness must additionally be traced. Section 5 briefly comments on this refinement. Finally, sections 6 and 7 discuss related work, future work and present the conclusions. For reasons of brevity and continuity, the formal proofs are not included in the paper, but can be found in [10].

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 [11] 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 $Atom$. 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 $\phi : Uvar \to Term$ 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 given 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 $\{u \mapsto \phi(u) \,|\, u \in dom(\phi)\}$. The identity mapping on $Uvar$ is called the empty substitution and is denoted by ϵ. Substitutions, sets of substitutions, and the set of all 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 respectively defined by: $\phi \upharpoonright U = \{u \mapsto \phi(u) \,|\, u \in dom(\phi) \cap U\}$ and $(\phi \circ \varphi)(u) = \phi(\varphi(u))$. 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 in terms of a predicate mgu. The predicate $mgu(E, \phi)$ 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: $mgu(E) = \{\phi \,|\, mgu(E, \phi)\}$ where*

$$
\begin{array}{rl}
mgu(\emptyset, \epsilon) & \\
mgu(v = v : E, \zeta) & if\, mgu(E, \zeta) \\
mgu(t = v : E, \zeta) & if\, mgu(v = t : E, \zeta) \\
mgu(v = t : E, \zeta \circ \eta) & if\, mgu(\eta(E), \zeta) \wedge v \notin var(t) \wedge \eta = \{v \mapsto t\} \\
mgu(f(t_1 \ldots t_n) = f(t'_1 \ldots t'_n) : E, \zeta) & if\, mgu(\{t_i = t'_i\}_{i=1}^n : E, \zeta)
\end{array}
$$

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 [4].

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 [4], 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 $\Upsilon : Pvar \to Rvar$.

Definition 2 ($unify$) *The partial mapping $unify : Atom \times Subst/\!\approx \times Atom \times Subst/\!\approx \to Subst/\!\approx$ is defined by: $unify(a, [\phi]_\approx, b, [\psi]_\approx) = [(\varphi \circ \phi) \upharpoonright Pvar]_\approx$ where $\varphi \in mgu(\{\phi(a) = \Upsilon(\psi(b))\})$.*

2.3 Sub-terms and paths

To reason about sharing and freeness to depth-k it is necessary to introduce some notation to identify the sub-terms of a term. Finite sequences of integers, paths, are used to distinguish the different occurrences of a sub-term within a term. Formally, the set of paths, $(p, q, r \in P \subseteq)$ $Path$, is defined to be the least set such that: $\lambda \in Path$ and $n{\cdot}p \in Path$ if $p \in Path$ and $n \in \mathcal{N}$ $= \{1, 2, \ldots\}$ (for n less or equal to the maximum arity of $Func$). Each sub-term, and therefore each variable occurrence, can be identified by a path, which navigates the way from the root of the term, to the sub-term. For instance, the paths $2{\cdot}1{\cdot}\lambda$, $1{\cdot}\lambda$, λ respectively identify the v, u and $f(u, g(v))$ sub-terms of $f(u, g(v))$.

The set of valid paths for a term t is denoted by $path(t)$ where $path(v) = \{\lambda\}$ if $v \in Uvar$; otherwise $path(f(t_1 \ldots t_n)) = \{\lambda\} \cup \{i{\cdot}p_i \mid p_i \in path(t_i) \wedge 1 \leq i \leq n\}$. It is convenient to regard $\cdot$ as concatenation and thus $(1 \cdot 2 \cdot \lambda) \cdot (3 \cdot \lambda) = 1 \cdot 2 \cdot 3 \cdot \lambda$. Formally the sub-term of t at p is denoted by $term_p(t)$, that is, $term_\lambda(t) = t$ and $term_{i \cdot p}(f(t_1 \ldots t_n)) = term_p(t_i)$. Hence, $term_{2\cdot 1\cdot\lambda}(f(u, g(v))) = v$, whereas $term_{1\cdot\lambda}(f(u, g(v))) = u$, and $term_\lambda(f(u, g(v))) = f(u, g(v))$. The mapping $term_p(t)$ is partial since it is only defined for $p \in path(t)$.

3 Depth-∞ framework for sharing

Abstract interpretation can provide focus for developing an analysis by emphasising the importance of abstracting data and illuminating the relationship between data, operations, and their abstract counterparts. In section 3.1, an abstraction for substitutions, the data, is proposed which represents structural properties of substitutions to arbitrary depth. Section 3.2, on the other hand, is devoted to defining a procedure for abstracting $unify$ to arbitrary depth, denoted depth-∞.

3.1 Abstracting substitutions to depth-∞

An abstract substitution is structured as a set of sharing groups where a sharing group is a (possibly empty) set of program variable and path pairs.

Definition 3 **(Occ_{Svar})** *The set of sharing groups, Occ_{Svar} is defined by: $Occ_{Svar} = \wp(Svar \times Path)$.*

$Svar$ 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 the positions of the shared variable, that is, where the shared variable occurs in the terms to which the program variables are bound. $Svar$ usually corresponds to $Pvar$. It is necessary to parameterise Occ, however, so that abstract substitutions are well-defined under renaming by Υ. The precise notion of abstraction is first defined for a substitution via *type* and then lifted to sets of substitutions.

Definition 4 (*occ* **and** *type*) *The mappings* $occ : Uvar \times Subst \to Occ_{Svar}$ *and* $type : Subst/\!\approx\, \to \wp(Occ_{Svar})$ *are defined by:* $occ(u, \phi) = \{\langle v, p\rangle \mid u = term_p(\phi(v)) \wedge v \in Svar\}$ *and* $type([\phi]_\approx) = \{occ(u, \phi) \mid u \in Uvar\}$.

The mapping *type* is well-defined since $type([\phi]_\approx) = type([\varphi]_\approx)$ 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).

Example 1 *Suppose* $Svar = \{v, w, x, y, z\}$ *and* $\phi = \{v \mapsto f,\ w \mapsto u,\ x \mapsto u,\ y \mapsto u',\ z \mapsto g(u, u', u')\}$. *The variables that occur through* $Svar$ *are* u *and* u'. *The variable* u *occurs in* w, x *and* z: *in* w *at position* λ; *in* x *at position* λ; *and in* z *at position* $1\cdot\lambda$. *Thus* u *defines the sharing group* $occ(u, \phi) = \{\langle w, \lambda\rangle, \langle x, \lambda\rangle, \langle z, 1\cdot\lambda\rangle\}$. *Similarly, the variable* u' *yields the sharing group* $occ(u', \phi) = \{\langle y, \lambda\rangle, \langle z, 2\cdot\lambda\rangle, \langle z, 3\cdot\lambda\rangle\}$. *Note that* $occ(v, \phi) = \ldots = occ(z, \phi) = \emptyset$, *and more generally,* $\emptyset \in type([\phi]_\approx)$ *for arbitrary* ϕ *since the codomain of a substitution is always finite. Thus the abstraction for* ϕ *is given by* $type([\phi]_\approx) = \{occ(u, \phi),\ occ(u', \phi),\ \emptyset\}$.

The abstraction *type* is analogous to the abstraction $\mathcal{A}$ used in [13] and implicit in [9]. Both abstractions are formulated in terms of sharing groups. The crucial difference is that *type*, as well as expressing the presence of a shared variable, additionally represents the position of the shared variable in the terms to which the program variables are bound.

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

Definition 5 ($Subst^*_{Svar}$) *The set of abstract substitutions,* $Subst^*_{Svar}$, *is defined by:* $Subst^*_{Svar} = \wp(Occ_{Svar})$.

As before [9, 13], $Subst^*_{Svar}$ $(\subseteq)$ is a complete lattice with set union as the lub. Unlike before, however, the finiteness of $Svar$ is not enough to ensure the finiteness of $Subst^*_{Svar}$. The *type* abstraction extends to sets of substitutions as follows.

Definition 6 (α_{type} **and** γ_{type}) *The abstraction and concretisation mappings* $\alpha_{type} : \wp(Subst/\!\approx) \to Subst^*_{Svar}$ *and* $\gamma_{type} : Subst^*_{Svar} \to \wp(Subst/\!\approx)$ *are defined by:* $\alpha_{type}(\Phi) = \cup_{[\phi]_\approx \in \Phi} type([\phi]_\approx)$ *and* $\gamma_{type}(\phi^*) = \{[\phi]_\approx \in Subst/\!\approx\, \mid type([\phi]_\approx) \subseteq \phi^*\}$.

The abstraction of a set of substitutions Φ merely combines all the sharing information from all the substitutions in Φ. The mappings α_{type} and γ_{type}

are monotonic. Note that $\alpha_{type}(\emptyset) = \emptyset$ whereas $\alpha_{type}(\Phi) = \{\emptyset\}$ if Φ is a set of substitutions which all ground $Svar$. The bottom element of $Subst^*_{Svar}$ $(\subseteq)$ is meaningful and, in fact, represents failure.

Abstract substitutions inherit their simple lub and their ability to propagate groundness because, like in [9, 13], the domain is formulated in terms of sharing groups. Examples 2 and 3 illustrate the lub and the expressiveness which comes from encoding structural properties of substitutions.

Example 2 *Suppose* $\mu = \{u \mapsto f(x, g(x))\}$, $\nu = \{u \mapsto f(x, g(x)),\ v \mapsto f(w, y)\}$ *and* $Svar = \{u,\ v,\ w,\ x,\ y\}$. *Then* $type([\mu]_\approx) = \mu^*$ *and* $type([\nu]_\approx) = \nu^*$ *where* $\mu^* = \{\{\langle v, \lambda\rangle\},\ \{\langle w, \lambda\rangle\},\ \{\langle u, 1{\cdot}\lambda\rangle,\ \langle u, 2{\cdot}1{\cdot}\lambda\rangle,\ \langle x, \lambda\rangle\},\ \{\langle y, \lambda\rangle\},\ \emptyset\}$ *and* $\nu^* = \{\{\langle v, 1{\cdot}\lambda\rangle,\ \langle w, \lambda\rangle\},\ \{\langle u, 1{\cdot}\lambda\rangle,\ \langle u, 2{\cdot}1{\cdot}\lambda\rangle,\ \langle x, \lambda\rangle\},\ \{\langle v, 2{\cdot}\lambda\rangle,\ \langle y, \lambda\rangle\},\ \emptyset\}$. *Observe that* $[\mu]_\approx,\ [\nu]_\approx \in \gamma_{type}(\mu^* \cup \nu^*)$.

Example 3 *Returning to* ϕ *of example 1, let* $\phi^* = type([\phi]_\approx) = \{\{\langle w, \lambda\rangle,\ \langle x, \lambda\rangle,\ \langle z, 1\cdot\lambda\rangle\},\ \{\langle y, \lambda\rangle,\ \langle z, 2\cdot\lambda\rangle,\ \langle z, 3\cdot\lambda\rangle\},\ \emptyset\}$. ϕ^* *can be interpreted as follows. The variables of* $Svar$ *which* ϕ *grounds, do not appear in* ϕ^**; and the variables of* $Svar$ *which are independent (unaliased), never occur in the same sharing group of* ϕ^*. *Thus* ϕ^* *represents that* v *is ground and that* x *and* y *are independent. Additionally,* ϕ^* *captures the fact that grounding either* x *or* w *grounds the other. Also* ϕ^* *indicates that* w, x *and* y *are possibly free whereas* z *is non-free [12]. It also shows that grounding* w, x *or the variable at the first argument of the term* $\phi(z)$, *grounds the others.*

3.2 Abstracting unification to depth-∞

The abstract $unify$ operator, $unify^*$, is defined by mimicking the unification algorithm, and just as $unify$ is defined in terms of mgu, $unify^*$ is formulated in terms of an abstraction of mgu, mge. 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 $v = t$. These simplified equations are then solved. The abstract equation solver mge adopts a similar strategy, but relegates the solution of the simplified equations to $solve$. (To be precise, mge abstracts a slight generalisation of mgu. Specifically, if $\varphi \in mgu(\phi(E))$ and $mge(E, type([\phi]_\approx), \psi^*)$ then $[\varphi \circ \phi]_\approx \in \gamma_{type}(\psi^*)$. The generalisation is convenient because it spares the need to define an extra (composition) operator for abstract substitutions.)

Definition 7 **(**mge**)** *The relation* $mge : Eqn \times Subst^*_{Svar} \times Subst^*_{Svar}$ *is defined by:*

$$\begin{array}{rl}
mge(\emptyset, \phi^*, \phi^*) & \\
mge(v = v : E, \phi^*, \psi^*) & \textit{if}\ mge(E, \phi^*, \psi^*) \\
mge(v = t : E, \phi^*, \psi^*) & \textit{if}\ mge(E, solve(v, t, \phi^*), \psi^*) \wedge \\
& \quad v \notin var(t) \\
mge(t = v : E, \phi^*, \psi^*) & \textit{if}\ mge(v = t : E, \phi^*, \psi^*) \\
mge(f(t_1 \ldots t_n) = f(t'_1 \ldots t'_n) : E, \phi^*, \psi^*) & \textit{if}\ mge(\{t_i = t'_i\}_{i=1}^n : E, \phi^*, \psi^*)
\end{array}$$

To define *solve*, and thereby *mge*, two auxiliary operators are required. The first, denoted $rel(t, \phi^*)$, calculates the sharing groups of ϕ^* which are relevant to the term t, that is, those sharing groups of ϕ^* which share variables with t. This is analogous to the *rel* operator of [9]. The second operator, $scale(o, P)$, denotes the sharing group formed by binding a variable to a non-ground term. The intuition behind *scale* is that if o ($\in Occ_{Svar}$) is the sharing group for a certain shared variable, and the variable is subsequently bound to a non-ground term containing a variable at p, then the sharing group for the new variable includes the sharing group $scale(o, \{p\})$. Definition 8 formally defines *rel* and *scale* and examples 4 and 5 demonstrate their use.

Definition 8 (*rel* and *scale*) *The mappings* $rel : Term \times Subst^*_{Svar} \rightarrow Subst^*_{Svar}$ *and* $scale : Occ_{Svar} \times \wp(Path) \rightarrow Occ_{Svar}$ *are defined by:* $rel(t, \phi^*) = \{o \in \phi^* \mid var(o) \cap var(t) \neq \emptyset\}$ *and* $scale(o, P) = \{\langle u, p_u \cdot p_P \rangle \mid \langle u, p_u \rangle \in o \wedge p_P \in P\}$.

Example 4 *Adopting* μ^* *and* ν^* *from example 2 and denoting* $\phi^* = \mu^* \cup \nu^*$, $rel(u, \phi^*) = \{\{\langle u, 1 \cdot \lambda \rangle, \langle u, 2 \cdot 1 \cdot \lambda \rangle, \langle x, \lambda \rangle\}\}$ *and* $rel(v, \phi^*) = \{\{\langle v, \lambda \rangle\}, \{\langle v, 1 \cdot \lambda \rangle, \langle w, \lambda \rangle\}, \{\langle v, 2 \cdot \lambda \rangle, \langle y, \lambda \rangle\}\}$.

Example 5 *Using* ϕ *of example 1, if* $\varphi = \{u \mapsto h(u')\}$ *then* $\varphi \circ \phi = \{v \mapsto f, w \mapsto h(u'), x \mapsto h(u'), y \mapsto u', z \mapsto g(h(u'), u', u')\}$ *and therefore* $type([\varphi \circ \phi]_{\approx}) = \{occ(u', \varphi \circ \phi), \emptyset\} = \{\{\langle w, 1 \cdot \lambda \rangle, \langle x, 1 \cdot \lambda \rangle, \langle y, \lambda \rangle, \langle z, 1 \cdot 1 \cdot \lambda \rangle, \langle z, 2 \cdot \lambda \rangle, \langle z, 3 \cdot \lambda \rangle\}, \emptyset\}$. *Note that* $scale(occ(u, \phi), \{1 \cdot \lambda\}) = \{\langle w, 1 \cdot \lambda \rangle, \langle x, 1 \cdot \lambda \rangle, \langle z, 1 \cdot 1 \cdot \lambda \rangle\}$ *which corresponds to the subset of* $occ(u', \varphi \circ \phi)$ *induced by* φ *binding* u *to* $h(u')$.

The nub of the equation solver 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^*. *solve* is formulated in terms of the fixed-point of *close*. The recursive definition of *close* generalises to the closure under union operation of [9, 13] and models the propagation of the aliases which arise during the solution of $v = t$. The full definition of *solve* is given below in definition 9. In definition 9, the notation $S \triangle S'$ denotes $(S \setminus S') \cup (S' \setminus S)$, the symmetric set difference of two sets S and S'.

Definition 9 (*solve*, *close* and *extend*) *The mappings* $solve : Svar \times Term \times Subst^*_{Svar} \rightarrow Subst^*_{Svar}$, $close : Svar \times Term \times Subst^*_{Svar} \rightarrow Subst^*_{Svar}$, $extend : Svar \times Term \times Occ_{Svar} \times Occ_{Svar} \rightarrow Subst^*_{Svar}$ *are defined by:*

$$solve(v, t, \phi^*) = lfp(close(v, t, \phi^*)) \setminus (rel(v, \phi^*) \triangle rel(t, \phi^*))$$

$$\begin{aligned}
close^0(v, t, \phi^*) &= \phi^* \\
close^{i+1}(v, t, \phi^*) &= \varphi^* \cup \left\{ o_{vt} \;\middle|\; \begin{array}{c} \varphi \in Subst \wedge o_v, o_t \in type([\varphi]_{\approx}) \cap \varphi^* \wedge \\ o_{vt} \in extend(v, t, o_v, o_t) \end{array} \right\} \\
&\quad \text{where } \varphi^* = close^i(v, t, \phi^*)
\end{aligned}$$

$$extend(v, t, o_v, o_t) = \\ \left\{ scale(o_v, S) \cup o_t \;\middle|\; \begin{array}{c} \langle v, p_v \rangle \in o_v \;\wedge\; s \in S \;\Leftrightarrow\; \langle v_t, p_t \rangle \in o_t \wedge \\ v_t = term_r(t) \;\wedge\; p_v \cdot s = r \cdot p_t \end{array} \right\} \cup \\ \left\{ o_v \cup scale(o_t, S) \;\middle|\; \begin{array}{c} \langle v_t, p_t \rangle \in o_t \wedge v_t = term_r(t) \wedge \\ s \in S \;\Leftrightarrow\; \langle v, p_v \rangle \in o_v \;\wedge\; p_v = r \cdot p_t \cdot s \end{array} \right\}$$

Example 6 *Consider again $\phi^* = \mu^* \cup \nu^*$ of example 4 and specifically the abstract substitution produced by solving the equation $v = u$ in the context of ϕ^*. For brevity let $\phi^* = \{o_1, \ldots, o_6, \emptyset\}$ where $o_1 = \{\langle v, \lambda \rangle\}$; $o_2 = \{\langle w, \lambda \rangle\}$; $o_3 = \{\langle u, 1 \cdot \lambda \rangle, \langle u, 2 \cdot 1 \cdot \lambda \rangle, \langle x, \lambda \rangle\}$; $o_4 = \{\langle y, \lambda \rangle\}$; $o_5 = \{\langle v, 1 \cdot \lambda \rangle, \langle w, \lambda \rangle\}$; and $o_6 = \{\langle v, 2 \cdot \lambda \rangle, \langle y, \lambda \rangle\}$. The close operator tracks the substitutions that can arise during the computation of a unifier in the unification algorithm. Sharing groups are iteratively combined until no more sharing groups can be generated and the fixed-point is reached.*

$$\begin{aligned} close(v, u, \phi^*) \;&=\; \phi^* \cup \{o_7, o_8, o_9\} \text{ where} \\ o_7 &= scale(o_1, \{1 \cdot \lambda, 2 \cdot 1 \cdot \lambda\}) \cup o_3 \\ &= \{\langle u, 1 \cdot \lambda \rangle, \langle u, 2 \cdot 1 \cdot \lambda \rangle, \langle v, 1 \cdot \lambda \rangle, \langle v, 2 \cdot 1 \cdot \lambda \rangle, \langle x, \lambda \rangle\} \\ o_8 &= o_3 \cup o_5 \\ &= \{\langle u, 1 \cdot \lambda \rangle, \langle u, 2 \cdot 1 \cdot \lambda \rangle, \langle v, 1 \cdot \lambda \rangle, \langle w, \lambda \rangle, \langle x, \lambda \rangle\} \\ o_9 &= scale(o_6, \{1 \cdot \lambda\}) \cup o_3 \\ &= \{\langle u, 1 \cdot \lambda \rangle, \langle u, 2 \cdot 1 \cdot \lambda \rangle, \langle v, 2 \cdot 1 \cdot \lambda \rangle, \langle y, 1 \cdot \lambda \rangle, \langle x, \lambda \rangle\} \end{aligned}$$

$$\begin{aligned} close(v, u, \phi^* \cup \{o_7, o_8, o_9\}) \;&=\; \phi^* \cup \{o_7, o_8, o_9, o_{10}\} \text{ where} \\ o_{10} &= scale(o_6, \{1 \cdot \lambda\}) \cup o_8 \;=\; o_5 \cup o_9 \\ &= \{\langle u, 1 \cdot \lambda \rangle, \langle u, 2 \cdot 1 \cdot \lambda \rangle, \langle v, 1 \cdot \lambda \rangle, \langle v, 2 \cdot 1 \cdot \lambda \rangle, \langle w, \lambda \rangle, \langle x, \lambda \rangle, \langle y, 1 \cdot \lambda \rangle\} \end{aligned}$$

$$close(v, u, \phi^* \cup \{o_7, o_8, o_9, o_{10}\}) \;=\; \phi^* \cup \{o_7, o_8, o_9, o_{10}\}$$

Each iteration of close combines a sharing group for a variable through v with a sharing group for a variable through u. In the case of o_7, for instance, the sharing groups o_1 and o_3 dictate the inclusion of the sharing group $o_7 = \{\langle u, 1 \cdot \lambda \rangle, \langle u, 2 \cdot 1 \cdot \lambda \rangle, \langle v, 1 \cdot \lambda \rangle, \langle v, 2 \cdot 1 \cdot \lambda \rangle, \langle x, \lambda \rangle\}$. This is because on unification of v and u, x will occur through v at positions $1 \cdot \lambda$ and $2 \cdot 1 \cdot \lambda$.

Note that sharing group o_{10} can be formed from either o_5 and o_9 or o_6 and o_8; and o_8 and o_9, in turn, are respectively derived from o_3 and o_5, and o_3 and o_6. The fact that o_{10} can be derived in two ways from o_3, o_5 and o_6 is a consequence of the non-determinism implicit in the unification algorithm. Note also that o_1 (and o_3) are barred from being combined with o_7 by virtue of the type check incorporated in close. In general, this check improves both the precision and analysis time by reducing the number of sharing groups that have to be combined. The underlying observation is that only consistent sharing groups need to be considered, that is, sharing groups that can share a common substitution. In the case of o_1 and o_7, for instance, no substitution can leave v free and bind v to a non-variable term. Thus o_1

and o_7 must characterise different substitutions. In fact, in this case, o_1 and o_7 correspond to substitutions which arise at different stages of the unification algorithm. Hence o_1 and o_7 never need to be combined.

From example 4, $rel(v, \phi^*) \triangle rel(u, \phi^*) = \{o_1, o_3, o_5, o_6\}$, and therefore it finally follows that $solve(v, u, \phi^*) = \{o_2, o_4, o_7, o_8, o_9, o_{10}, \emptyset\}$. The intuition behind $rel(v, \phi^*) \triangle rel(u, \phi^*)$ is that it represents those sharing groups for shared variables which pass through either v or u, but not both. After unification, any variable which passes through v must also pass through u and vice versa. Sharing groups which do not possess this property are redundant, and in fact represent grounded variables, and hence can be removed.

Theorem 3.1

$$\begin{aligned}&[\phi]_\approx \in \gamma_{type}(\phi^*) \wedge \varphi \in mgu(\phi(E)) \wedge \\ &\quad var(E) \subseteq Svar \wedge mge(E, \phi^*, \psi^*) \Rightarrow [\varphi \circ \phi]_\approx \in \gamma_{type}(\psi^*)\end{aligned}$$

It is convenient a shorthand to regard mge as a mapping, that is, $mge(E, \phi^*) = \psi^*$ if $mge(E, \phi^*, \psi^*)$. Strictly, it is necessary to show that $mge(E, \phi^*, \psi^*)$ is deterministic for $mge(E, \phi^*)$ to be well-defined. Like in [5], the conjecture is that mge yields a unique abstract substitution ψ^* for ϕ^* regardless of the order in which E is solved (though, in practice, any ψ^* is safe).

To approximate the $unify$ operation it is convenient to introduce a collecting semantics, concerned with sets of substitutions - the collecting domain, 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 10 ($unify^c$) *The mapping $unify^c : Atom \times \wp(Subst/\approx) \times Atom \times \wp(Subst/\approx) \rightarrow \wp(Subst/\approx)$ is defined by: $unify^c(a, \Phi, b, \Psi) = \{[\theta]_\approx \mid [\phi]_\approx \in \Phi \wedge [\psi]_\approx \in \Psi \wedge [\theta]_\approx = unify(a, [\phi]_\approx, b, [\psi]_\approx)\}$.*

The usefulness of the collecting semantics as a form of program analysis is negated by the fact that it can lead to non-terminating computations. The collecting semantics, however, is a useful tool for reasoning about the correctness of $unify^*$. To define $unify^*$ and prove safety it is necessary to introduce an abstract restriction operator, $\cdot \,\bar{}^{*}\, \cdot$, defined by: $\mu^* \,\bar{}^{*}\, U = \{o \,\bar{}^{*}\, U \mid o \in \mu^*\}$ and $o \,\bar{}^{*}\, U = \{\langle u, p\rangle \in o \mid u \in U\}$. The definition of $unify^*$ is given below and theorem 3.2 assumes $var(a) \cup var(b) \subseteq Pvar$.

Definition 11 ($unify^*$) *The mapping $unify^* : Atom \times Subst^*_{Pvar} \times Atom \times Subst^*_{Pvar} \rightarrow Subst^*_{Pvar}$ is defined by: $unify^*(a, \phi^*, b, \psi^*) = mge(\{a = \Upsilon(b)\}, \phi^* \cup \Upsilon(\psi^*)) \,\bar{}^{*}\, Pvar$.*

Theorem 3.2 (local safety of $unify^*$)

$$\begin{aligned}&\Phi \subseteq \gamma_{type}(\phi^*) \wedge \Psi \subseteq \gamma_{type}(\psi^*) \Rightarrow \\ &\quad unify^c(a, \Phi, b, \Psi) \subseteq \gamma_{type}(unify^*(a, \phi^*, b, \psi^*))\end{aligned}$$

$$
\begin{tikzcd}
Subst^{\#}_{Pvar} \arrow[r, "unify^{\#}"] & Subst^{\#}_{Pvar} \arrow[d, "\gamma_{type^*}"'] \arrow[dd, bend left=60, "\gamma_{type^{\#}}"] \\
Subst^{*}_{Pvar} \arrow[u, "\alpha_{type^*}"] \arrow[r, "unify^{*}"] & \overset{\subseteq}{Subst^{*}_{Pvar}} \arrow[d, "\gamma_{type}"'] \\
\wp(Subst/\approx) \arrow[u, "\alpha_{type}"] \arrow[uu, bend left=60, "\alpha_{type^{\#}}"] \arrow[r, "unify^{c}"] & \overset{\subseteq}{\wp(Subst/\approx)}
\end{tikzcd}
$$

Figure 1: The relationship between depth-∞ and depth-k abstractions.

4 Depth-k analysis for sharing

The usefulness of the depth-∞ framework is compromised by its infiniteness. Since $Path$ is not finite, $Subst^{*}_{Svar}(\subseteq)$ is not a finite lattice, and therefore a fixed-point computation, of the sort employed in [1, 13] does not automatically terminate. Section 4.1 explains how convergence of the iterates can be enforced by replacing $Subst^{*}_{Svar}$ $(\subseteq)$ with $Subst^{\#}_{Svar}$ $(\subseteq)$ and throttling paths to length k. The mappings α_{type^*} and γ_{type^*} formalise this depth-k abstraction. Section 4.1 describes an abstract analog of $unify^{*}$, $unify^{\#}$, which safely operates on truncated paths. Figure 1 illustrates that by cascading the depth-∞ and depth-k approximations end-to-end, $unify^{\#}$ can be regarded an abstraction of $unify^{c}$ via the abstraction and concretisation mappings $\alpha_{type^{\#}}$ and $\gamma_{type^{\#}}$. These two mappings are respectively defined by composing α_{type^*} with α_{type} and γ_{type} with γ_{type^*}. (The $^{\#}$ notation is used to distinguish depth-k objects from their depth-∞ counterparts.)

4.1 Abstracting depth-∞ abstract substitutions to depth-k

Depth-k abstractions [14, 17] normally represent the principal functor of a term complete with descriptors for its sub-terms. Sub-term descriptions are given to the depth of a predetermined constant bound k. (Depth-k analyses are conventionally good at representing structure but are often weak at tracing groundness and aliasing information.) In the spirit of the depth-k approach, the arbitrary length paths of the depth-∞ framework can be truncated to length k. (By discarding paths all together, the sharing analysis of [9] is obtained [10]!) Truncating at k induces an approximation and the notion of an abstract path $Path^{\#}_{\infty}$. $Path^{\#}_{\infty}$ thus includes a symbol nf to denote approximation and is formally defined to be the least set such that: $\lambda \in Path^{\#}_{\infty}$, $nf \in Path^{\#}_{\infty}$ and $n \cdot p^{\#} \in Path^{\#}_{\infty}$ if $p^{\#} \in Path^{\#}_{\infty}$ and $n \in \mathcal{N}$ (for n less or equal to the maximum arity of $Func$). The nf terminator can be interpreted as representing a set of paths. For example, $1{\cdot}2{\cdot}nf$, finitely represents the infinite set $\{(1{\cdot}2{\cdot}\lambda) \cdot p \,|\, p \in Path\}$. Note that $Path \subseteq Path^{\#}_{\infty}$.

Like before, $\cdot$ is interpreted as concatenation so that $(1{\cdot}2{\cdot}\lambda)\cdot(3{\cdot}nf) = 1{\cdot}2{\cdot}3{\cdot}nf$. It is also convenient (to simplify $\propto_{path}$ and $diff_\infty$) to let $(1\cdot nf)\cdot(2\cdot 3\cdot\lambda) = 1\cdot 2\cdot 3\cdot\lambda$ and $(1\cdot 2\cdot nf)\cdot(3\cdot nf) = 1\cdot 2\cdot 3\cdot nf$. Finiteness is introduced by the mapping $depth_k$.

Definition 12 *The mapping $depth_k : Path^\#_\infty \to Path^\#_\infty$ is defined by:*

$$depth_k(n_1\cdot\ldots\cdot n_l\cdot n) = \begin{cases} n_1\cdot\ldots\cdot n_l\cdot n & \textit{if } l \le k \\ n_1\cdot\ldots\cdot n_k\cdot nf & \textit{if } l > k \end{cases} \quad \textit{where } n_i \in \mathcal{N},\, n \in \{\lambda, nf\}$$

The codomain of $depth_k$ defines a finite set of truncated paths: $Path^\#_k = \{depth_k(p^\#) \mid p^\# \in Path^\#_\infty\}$. $Path^\#_k$ is finite because, for a given program, $Func$ is finite. The notion of approximation implicit in the set of abstract paths is encapsulated by the poset $Path^\#_\infty$ $(\propto_{path})$ defined as the least reflexive relation such that: $p\cdot nf \propto_{path} p\cdot p^\#$ if $p \in Path$ and $p^\# \in Path^\#_\infty$. Thus, for example, $1\cdot nf \propto_{path} 1\cdot 2\cdot\lambda$, $1\cdot nf \propto_{path} 1\cdot 2\cdot nf$, $\lambda \propto_{path} \lambda$ and $nf \propto_{path} nf$.

It is convenient to use abstract paths from both $Path^\#_k$ and $Path^\#_\infty$ in the analysis. For precision, the intermediate calculations of the abstract unification algorithm use $Path^\#_\infty$. For termination, the abstract unifiers are approximated (widened) by collapsing paths into $Path^\#_k$. $Path^\#_k$ and $Path^\#_\infty$ induce two notions of abstract sharing group: $Occ_{Svar,k}$ and $Occ_{Svar,\infty}$. $Occ_{Svar,k}$ is finite whereas $Occ_{Svar,\infty}$ is infinite. The abstract domain $Subst^\#_{Svar}$ is formulated in terms of $Occ_{Svar,\infty}$.

Definition 13 ($Occ^\#_{Svar,k}$ and $Occ^\#_{Svar,\infty}$) *The sets of (abstracted) sharing groups are defined by: $Occ^\#_{Svar,k} = \wp(Svar \times Path^\#_k)$ and $Occ^\#_{Svar,\infty} = \wp(Svar \times Path^\#_\infty)$.*

Definition 14 ($Subst^\#_{Svar}$) *The set of (abstracted) abstract substitutions, $Subst^\#_{Svar}$, is defined by: $Subst^\#_{Svar} = \wp(Occ^\#_{Svar,\infty})$.*

$Subst^\#_{Svar}$ has $\cup$ as its lub. The poset $\propto_{path}$ induces the preorders $Occ^\#_{Svar,k}$ $(\propto_{share})$ and $Occ^\#_{Svar,\infty}$ $(\propto_{share})$ which formalise approximation among abstract sharing groups. The preorders are defined by: $o^\# \propto_{share} o'^\#$ if and only if $var(o^\#) = var(o'^\#)$ and for all $\langle u, p'^\#\rangle \in o'^\#$ there exists $\langle u, p^\#\rangle \in o^\#$ such that $p^\# \propto_{path} p'^\#$. Note that $\propto_{share}$ formalises approximation but does not specify how to threshold a sharing group. The operator $share_k$ is thus introduced to perform thresholding.

Definition 15 ($share_k$) *The mappings $share_k : Occ^\#_{Svar,\infty} \to Occ^\#_{Svar,k}$ is defined by: $share_k(o) = \{\langle u, depth_k(p)\rangle \mid \langle u, p\rangle \in o\}$.*

Since $Occ_{Svar} \subseteq Occ^\#_{Svar,\infty}$, $share_k$ can threshold sharing groups as well as abstract sharing groups. Thus α_{type*} can be defined in terms of $share_k$ whereas the concretisation γ_{type*} can be formulated in terms of $\propto_{share}$.

Definition 16 (α_{type^*} **and** γ_{type^*}) *The mappings* $\alpha_{type^*} : \wp(Subst^*_{Svar}) \to Subst^{\#}_{Svar}$ *and* $\gamma_{type^*} : Subst^{\#}_{Svar} \to \wp(Subst^*_{Svar})$ *are defined by:* $\alpha_{type^*}(\Phi^*) = \cup_{\phi^* \in \Phi^*}\{share_k(o) \mid o \in \phi^*\}$ *and* $\gamma_{type^*}(\phi^{\#}) = \{\phi^* \in Subst^*_{Svar} \mid \forall o \in \phi^* . \exists o^{\#} \in \phi^{\#} . o^{\#} \propto_{share} o\}$.

Finally, the link between the collecting domain $\wp(Subst/\approx)$ and $Subst^{\#}_{Svar}$ can be made explicit by cascading α_{type^*} with α_{type} and γ_{type} with γ_{type^*}.

Definition 17 ($\alpha_{type^{\#}}$ **and** $\gamma_{type^{\#}}$) *The mappings* $\alpha_{type^{\#}} : \wp(Subst/\approx) \to Subst^{\#}_{Svar}$ *and* $\gamma_{type^{\#}} : Subst^{\#}_{Svar} \to \wp(Subst/\approx)$ *are defined by:* $\alpha_{type^{\#}}(\Phi) = \alpha_{type^*}(\{\alpha_{type}(\Phi)\})$ *and* $\gamma_{type^{\#}}(\Phi^{\#}) = \cup_{\phi^* \in \gamma_{type^*}(\Phi^{\#})}\gamma_{type}(\phi^*)$.

To strike an analogy with conventional sharing groups [9], it is insightful to introduce depth-k versions of occ and $type$, namely $occ^{\#}_k$ and $type^{\#}_k$.

Definition 18 ($occ^{\#}_k$ **and** $type^{\#}_k$) *The mappings* $occ^{\#}_k : Uvar \times Subst \to Occ^{\#}_{Svar,k}$ *and* $type^{\#}_k : Subst/\approx \to \wp(Occ^{\#}_{Svar,k})$ *are defined by:* $occ^{\#}_k(u, \phi) = \{\langle v, depth_k(p)\rangle \mid u = term_p(\phi(v)) \wedge v \in Svar\}$ *and* $type^{\#}_k([\phi]_\approx) = \{occ^{\#}_k(u, \phi) \mid u \in Uvar\}$.

If $depth_\infty$ is regarded as the identity mapping on $Path^{\#}_\infty$, then $occ^{\#}_\infty = occ$ and $type^{\#}_\infty = type$, and lemma 4.1 immediately follows. Lemma 4.1 succinctly expresses $\alpha_{type^{\#}}$ and $\gamma_{type^{\#}}$ in a familiar format.

Lemma 4.1 $\alpha_{type^{\#}}(\Phi) = \cup_{[\phi]_\approx \in \Phi}\, type^{\#}_k([\phi]_\approx)$ *and* $\gamma_{type^{\#}}(\phi^{\#}) = \{[\phi]_\approx \in Subst/\approx \mid \forall o^{\#} \in type^{\#}_\infty([\phi]_\approx) . \exists\, o'^{\#} \in \phi^{\#} . o'^{\#} \propto_{share} o^{\#}\}$.

4.2 Abstracting depth-∞ abstract unification to depth-k

A truncated path version of $unify^*$, $unify^{\#}$, is constructed by defining a depth-k analog of mge. The analog, denoted $mge^{\#}_\infty \subseteq Eqn \times Subst^{\#}_{Svar} \times Subst^{\#}_{Svar}$, simplifies and solves syntactic equations in the style of mge in the obvious way with the exception that $solve$ is replaced with $solve^{\#}_\infty$. The operator $solve^{\#}_\infty$ solves a syntactic equation of the form $v = t$ in the presence of the abstract substitution $\phi^{\#}$, returning the abstract unifier composed with $\phi^{\#}$. Thus $solve^{\#}_\infty$ abstracts $solve$ and, like before, is the key element of $mge^{\#}_\infty$.

Concatenation implicitly defines a notion of difference and to flesh out $solve^{\#}_\infty$, it is necessary to introduce an abstract difference operator, $diff_\infty$.

Definition 19 ($diff_\infty$) *The partial mapping* $diff_\infty : Path^{\#}_\infty \times Path^{\#}_\infty \to Path^{\#}_\infty$ *is defined by:*

$$diff_\infty(p^{\#}, r^{\#}) = \begin{cases} q & \text{if } p^{\#} \in Path \wedge r^{\#} \in Path \wedge \quad p^{\#} \cdot q = r^{\#} \\ nf & \text{if } p^{\#} \notin Path \wedge r^{\#} \in Path \wedge \quad p^{\#} \cdot s = r^{\#} \\ nf & \text{if } p^{\#} \in Path \wedge r^{\#} \notin Path \wedge \qquad p^{\#} = r^{\#} \cdot s \\ q^{\#} & \text{if } p^{\#} \in Path \wedge r^{\#} \notin Path \wedge p^{\#} \cdot q^{\#} = r^{\#} \\ nf & \text{if } p^{\#} \notin Path \wedge r^{\#} \notin Path \end{cases}$$

Lemma 4.2 states precisely how $diff_\infty$ relates to $\cdot$.

Lemma 4.2 $p^\# \propto_{path} p \wedge r^\# \propto_{path} r \wedge p{\cdot}q = r \Rightarrow diff_\infty(p^\#, r^\#) \propto_{path} q$

Additionally, an auxiliary operator $scale^\#_\infty$ is required to abstract $scale$.

Definition 20 **(**$scale^\#_\infty$**)** *The mapping scale* $: Occ_{Svar,\infty} \times \wp(Path^\#_\infty) \to Occ_{Svar,\infty}$ *is defined by:*

$$scale^\#_\infty(o^\#, S^\#) = \left\{ \langle u, p^\# \cdot s^\# \rangle \;\middle|\; \begin{array}{c} \langle u, p^\# \rangle \in o^\# \wedge \\ p^\# \in Path \wedge s^\# \in S^\# \end{array} \right\} \cup \left\{ \langle u, p^\# \rangle \;\middle|\; \begin{array}{c} \langle u, p^\# \rangle \in o^\# \wedge \\ p^\# \notin Path \end{array} \right\}$$

With $rel^\#$, the analog of rel for $Subst^\#_{Svar}$, $solve^\#_\infty$ and $close^\#_\infty$ can be constructed by plugging the appropriate depth-k operators into $solve$ and $close$.

Definition 21 **(**$solve^\#_\infty$**,** $close^\#_\infty$ **and** $extend^\#_\infty$**)** *The mappings* $solve^\#_\infty : Svar \times Term \times Subst^\#_{Svar} \to Subst^\#_{Svar}$, $close^\#_\infty : Svar \times Term \times Subst^\#_{Svar} \to Subst^\#_{Svar}$, $extend^\#_\infty : Svar \times Term \times Occ_{Svar,\infty} \times Occ_{Svar,\infty} \to Subst^\#_{Svar}$ *are defined by:*

$$solve^\#_\infty(v, t, \phi^\#) = lfp(close^\#_\infty(v, t, \phi^\#)) \setminus (rel^\#(v, \phi^\#) \triangle rel^\#(t, \phi^\#))$$

$$\begin{array}{rl} close^{\#\,0}_\infty(v, t, \phi^*) = & \phi^\# \\ close^{\#\,i+1}_\infty(v, t, \phi^*) = & \varphi^\# \cup \left\{ o^\#_{vt} \;\middle|\; o^\#_v, o^\#_t \in \varphi^\# \wedge o^\#_{vt} \in extend^\#_\infty(v, t, o^\#_v, o^\#_t) \right\} \\ & \text{where } \varphi^\# = close^{\#\,i}_\infty(v, t, \phi^\#) \end{array}$$

$$extend^\#_\infty(v, t, o^\#_v, o^\#_t) = \\ \left\{ scale^\#_\infty(o^\#_v, S^\#) \cup o^\#_t \;\middle|\; \begin{array}{c} \langle v, p^\#_v \rangle \in o^\#_v \;\wedge\; s^\# \in S^\# \;\Leftrightarrow\; \langle v_t, p^\#_t \rangle \in o^\#_t \;\wedge \\ v_t = term_r(t) \;\wedge\; diff_\infty(p^\#_v, r \cdot p^\#_t) = s^\# \end{array} \right\} \cup \\ \left\{ o^\#_v \cup scale^\#_\infty(o^\#_t, S^\#) \;\middle|\; \begin{array}{c} \langle v_t, p^\#_t \rangle \in o^\#_t \;\wedge\; v_t = term_r(t) \;\wedge\; s^\# \in S^\# \;\Leftrightarrow \\ \langle v, p^\#_v \rangle \in o^\#_v \;\wedge\; diff_\infty(r \cdot p^\#_t, p^\#_v) = s^\# \end{array} \right\}$$

Note that $solve^\#_\infty$ and each of its constituent parts are independent of k. Thus $solve^\#_\infty$ is an abstract equation solver for depth-k abstractions of arbitrary k. Precision is throttled (without touching the core components of the analysis) at the level of $mge^\#_\infty$. Specifically, an intermediate construction, $mge^\#_k$, is employed to threshold the abstract unifier to depth-k.

Definition 22 **(**$mge^\#_k$**)** *The relation* $mge^\#_k : Eqn \times Subst^\#_{Svar} \times Subst^\#_{Svar}$ *is defined by:* $mge^\#_k(E, \phi^\#, \varphi^\#)$ *if* $mge^\#_\infty(E, \phi^\#, \psi^\#)$ *where* $\varphi^\# = \{share_k(o^\#) \mid o^\# \in \psi^\#\}$.

Like before, it is convenient to regard $mge_k^\#$ as a mapping. Then, with the addition of some renaming machinery, $mge_k^\#$ defines a depth-k version of $unify^*$, $unify_k^\#$. Safety is stated as theorem 4.3 and is couched in terms of $unify^c$. Like theorem 3.2, theorem 4.3 assumes $var(a) \cup var(b) \subseteq Pvar$. Also $^{-\#}$ is the obvious depth-k analog of restriction.

Definition 23 ($unify^\#$) *The mapping* $unify^\# : Atom \times Subst_{Pvar}^\# \times Atom \times Subst_{Pvar}^\# \to Subst_{Pvar}^\#$ *is defined by:* $unify^\#(a, \phi^\#, b, \psi^\#) = mge_k^\#(\{a = \Upsilon(b)\}, \phi^\# \cup \Upsilon(\psi^\#)) \;^{-\#}\; Pvar$.

Theorem 4.3 (local safety of $unify^\#$**)**

$$\begin{aligned} &\Phi \subseteq \gamma_{type^\#}(\phi^\#) \;\wedge\; \Psi \subseteq \gamma_{type^\#}(\psi^\#) \;\Rightarrow \\ &\quad unify^c(a, \Phi, b, \Psi) \subseteq \gamma_{type^\#}(unify^\#(a, \phi^\#, b, \psi^\#)) \end{aligned}$$

5 Depth-∞ and depth-k freeness

The applications domain of the analysis can be enriched by augmenting the domains $Subst_{Svar}^*$ and $Subst_{Svar}^\#$ with a definite freeness component. Although $Subst_{Svar}^*$ and $Subst_{Svar}^\#$ succinctly express possible freeness and definite groundness, they cannot adequately record definite freeness. For instance, if $\Phi = \{\{u \mapsto f\}, \{u \mapsto v\}\}$ and $Svar = \{u, v\}$ then $\alpha_{type}(\Phi) = \{\{\langle u, \lambda\rangle, \langle v, \lambda\rangle\}, \emptyset\}$, and information about the non-freeness of u is lost. By additionally recording possible groundness, however, definite freeness can be inferred. For example, by adopting a domain $Subst_{Svar}^* \times Grnd_{Svar}^*$ in which $Grnd_{Svar}^* = \wp(Svar \times Path)$, then Φ could be represented as $\langle\{\{\langle u, \lambda\rangle, \langle v, \lambda\rangle\}, \emptyset\}, \{\langle u, \lambda\rangle\}\rangle$ indicating that a sub-term $term_\lambda(\phi(u))$ is ground for some $\phi \in \Phi$. Conversely, without a pair $\langle u, p\rangle$, $term_\lambda(\phi(u))$ must definitely be free. Extending the depth-∞ and depth-k analyses in this way is straightforward (though technical) and reuses a lot of the (possible) freeness abstract interpretation machinery. For brevity, example 7 illustrates the basic idea behind the tracking of (possible and definite) freeness in sub-terms with an example adapted from a benchmark program.

Example 7 *Consider the head unification* $unify_1^\#(a,\ \phi^\#,\ b,\ \psi^\#)$ *where* $a = dfri(tree(L,\ R),\ -(LFriHead,\ RFriTail))$, $b = dfri(Tree,\ DFri)$, $\phi^\# = \{\{\langle L,\ \lambda\rangle\},\ \{\langle LFriHead,\ \lambda\rangle\},\ \{\langle R,\ \lambda\rangle\},\ \{\langle RFriHead,\ \lambda\rangle\},\ \{\langle RFriTail,\ \lambda\rangle\},\ \emptyset\}$ *and* $\psi^\# = \{\{\langle DFri,\ 1\cdot\lambda\rangle\},\ \emptyset\}$. *Supposing* $\Upsilon(Tree) = Tree'$ *and* $\Upsilon(DFri) = DFri'$, *then* $unify_1^\#(a,\ \phi^\#,\ b,\ \psi^\#) = mge_1^\#(E,\ \phi^\# \cup \Upsilon(\psi^\#)) \;^{-\#}\; Pvar$ *where* $E = \{Tree' = tree(L,\ R),\ DFri' = -(LFriHead,\ RFriTail)\}$ *and* $\phi^\# \cup \Upsilon(\psi^\#) = \{\{\langle L,\ \lambda\rangle\},\ \{\langle LFriHead,\ \lambda\rangle\},\ \{\langle R,\ \lambda\rangle\},\ \{\langle RFriHead,\ \lambda\rangle\},\ \{\langle RFriTail,\ \lambda\rangle\},\ \{\langle DFri',\ 1\cdot\lambda\rangle\},\ \emptyset\}$. *But* $mge_\infty^\#(E,\ \phi^\# \cup \Upsilon(\psi^\#)) = \{\{\langle DFri',\ 1\cdot\lambda\rangle,\ \langle LFriHead,\ \lambda\rangle\},\ \{\langle RFriHead,\ \lambda\rangle\},\ \emptyset\}$ *and thus* $unify_1^\#(a,\ \phi^\#,\ b,\ \psi^\#) = \{\{\langle LFriHead,\ \lambda\rangle\},\ \{\langle RFriHead,\ \lambda\rangle\},\ \emptyset\}$. *Hence, L and R are grounded by head unification whereas* $LFriHead$ *and* $RFriHead$ *remain* *(possibly) free and unaliased. If, in addition, the possible grounding analysis does not include any pairs $\langle LFriHead, p\rangle$ and $\langle RFriHead, p\rangle$, then the definite freeness of $LFriHead$ and $RFriHead$ immediately follows. Here, depth-1 analysis is vital for the required precision.*

6 Related and future work

Recently, three relevant proposals for computing sharing have been put forward in the literature. In the first proposal [7], multiple analyses are run in lock step. This paper likewise follows the trend for simultaneously tracing different properties (namely groundness, sharing and freeness), but instead explains how the restructuring of domains can yield a depth-k analysis which cannot be synthesised in terms of the combined domain approach.

In the second proposal [6], the correctness of sharing and definite freeness analyses are considered. An abstract unification algorithm is proposed as a basis for constructing accurate freeness analyses with a domain formulated in terms of a system of abstract equations. Safety follows because the abstract algorithm mimics the unification algorithm in an intuitive way. Correctness is argued likewise here. One essential distinction between the two works is that the approach proposed in this paper uses paths to encode more accurate sharing information than the abstract equations of [6].

Very recently, in the third proposal [2], an analysis for sharing, groundness, linearity and definite 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 glues the Jacobs and Langen [9] structure with a conventional notion of freeness. Freeness is not generalised to depth-k and is not embedded into sharing groups in the way that is described in this paper.

Future work will focus on incorporating linearity into sharing groups embellished with depth-k freeness. Benchmarking will quantitatively assess the usefulness and efficiency of this refinement, and suggest also suitable k.

7 Conclusions

A powerful and formally justified analysis has been presented for inferring groundness, freeness, and sharing between the variables of a logic program. The analysis elegantly represents freeness information in a sharing group format. By revising sharing groups to capture freeness, aliasing behaviour can be precisely captured; groundness information can be accurately propagated; and in addition, the freeness of sub-terms can be tracked.

References

[1] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *J. of 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] 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, 1991.

[4] J. Lassez *et al.* *Foundations of Deductive Databases and Logic Programming*, chapter Unification Revisited. Morgan Kaufmann, 1987.

[5] M. Codish *et al.* Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In *ICLP'91*, pages 79–93, Paris, 1991. MIT Press.

[6] M. Codish *et al.* Freeness analysis for logic programs - and correctness? In *ICLP'93*, pages 116–131. MIT Press, June 1993.

[7] M. Codish *et al.* Improving abstract interpretation by combining domains. In *PEPM'93*. ACM Press, 1993.

[8] M. Hermenegildo and F. Rossi. Non-strict independent and-parallelism. In *ICLP'90*, pages 237–252, Jerusalem, 1990. MIT Press.

[9] D. Jacobs and A. Langen. Static Analysis of Logic Programs. *J. of Logic Programming*, pages 154–314, 1992.

[10] A. King. Depth-k sharing and freeness. Technical Report CSTR 93-14, Southampton University, S09 5NH, UK, 1993.

[11] J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.

[12] K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In *ICLP'91*, pages 49–63, Paris, 1991. The MIT Press.

[13] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency through abstract interpretation. *J. of Logic Programming*, pages 315–437, 1992.

[14] T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.

[15] H. Søndergaard. An application of the abstract interpretation of logic programs: occur-check reduction. In *ESOP'86*, pages 327–338, 1986.

[16] R. Sundararajan and J. Conery. An abstract interpretation scheme for groundness, freeness, and sharing analysis of logic programs. In *12th FST and TCS Conference*, New Delhi, December 1992.

[17] A. Taylor. *High Performance Prolog Implementation.* PhD thesis, Basser Department of Computer Science, July 1991.