Type Analysis of Logic Programs in the Presence of Type Definitions

Lunjin Lu
School of Computer Science
The University of Birmingham
Edgbaston, Birmingham B15 2TT, U.K.
L.Lu@cs.bham.ac.uk

Abstract

This paper presents an abstract domain and an abstract unification function for type analysis of logic programs with type definitions. Type information is inferred together with sharing and aliasing information. Aliasing information is used to improve the precision of type analysis.

1 Introduction

Prolog is a type-free language. The programmer does not have to specify the types of variables, functions and predicates. This may make it simple to write simple programs. However, it also makes it difficult to debug programs because type errors cannot be detected by Prolog systems. A type error will manifest itself in the form of a wrong result or a missing result rather than an indication of a type violation.

There have been many efforts to augment Prolog with type systems in the forms of type checking systems [2, 11, 27] and type analysis systems. Type analysis systems infer types from the text of a program. Some type analysis systems infer a type for the program and the type is meant to denote a superset of the success set of the program [1, 10, 12, 13, 25, 29, 32, 33, 34]. Other type analysis systems infer types for procedure calls and successes [4, 14, 16, 19, 20].

There are three main approaches for inferring types [13]. The first approach obtains from a program a system of formulae which describes set-theoretic relationships between the atoms appearing in the program. A type is then defined to be a distinguished model of the system of formulae. Examples of this approach are to be found in [1, 10, 25, 29, 32, 33, 34]. The second approach uses an abstract immediate consequence operator to approximate the immediate consequence operator $T_{\mathcal{P}}$ associated with the program $\mathcal{P}$ [31]. A type is defined as a fixed-point of the abstract immediate consequence operator. An example of this approach appears in [33]. Heintze and Jaffar [12, 13] show that the first and the second approaches are equivalent in the sense that a type in one approach has its counterpart in the other. The third approach involves specialising an abstract interpretation framework by developing abstract domains and their associated abstract functions. Some examples are to be found in [4, 14, 16, 19, 20].

Most type analysis systems infer descriptive types. A descriptive type is a description of a set of terms by an expression in a type language. Different type analysis systems may use different type languages. Most of them use regular types or deterministic regular types [10] in one form or another. A descriptive type is not intuitive. Although the sets of terms described by descriptive types are well-defined, it is difficult for the programmer to understand descriptive types. Horiuchi and Kanamori and Kawamura [14, 19, 20] infer prescriptive types where the types of function symbols are provided by the programmer.

This paper reports our work in inferring prescriptive types by means of abstract interpretation. The novelties of our work include (i) inferring prescriptive types together with sharing and aliasing information, and (ii) using aliasing information to improve the precision of type analysis. By aliasing, we refer to the situation where two variables are instantiated to syntactically identical terms.

Much work has been done on abstract interpretation of logic programs. For a comprehensive survey, see [9]. A number of frameworks have been brought about for abstract interpretation of logic programs [3, 18, 23, 24]. Specialising a framework for a particular analysis involves devising an abstract domain for descriptions of sets of substitutions and designing a number of abstract functions. One important abstract function is abstract unification [6, 17].

This paper puts forward an abstract domain and an abstract unification function for inferring type information together with sharing and aliasing information. The remainder of this paper is organised as follows. Section 2 briefly recalls some concepts and notations in logic programming and abstract interpretation, and introduces the problem of abstract unification. Section 3 proposes an abstract domain for type analysis and section 4 presents some lemmas and ancillary functions which are used in our abstract unification function. Section 5 gives our abstract unification function and proves its safety. Section 6 compares our work with other work and concludes the paper.

2 Preliminaries

We assume that the reader is familiar with terminology of logic programming [21]. Let Σ be a set of *function symbols*, Π be a set of *predicate symbols*, $\mathcal{VAR}$ be a denumerable set of variables. $\mathcal{TERMS}$ denotes the set of *terms* that can be constructed from Σ and $\mathcal{VAR}$. $\mathcal{ATOMS}$ denotes the set of *atoms* constructible from Π, Σ and $\mathcal{VAR}$.

$dom(\theta)$ denotes the domain of a *substitution* θ and ϵ denotes the *identity substitution*. An *expression* is a term, an atom, a literal, a clause, a goal etc. $vars(E)$ denotes the set of variables in E and $E\theta$ denotes the *instance* of E under θ. An expression E' is an instance of another expression E if $E' \equiv E\theta$ for some substitution θ where $A \equiv B$ denotes that A is syntactically identical to B. $\sigma \circ \theta$ denotes *composition* of two substitutions σ and θ.

Let θ be a substitution and $\mathcal{V} \subseteq \mathcal{VAR}$. $\theta\!\restriction\!\mathcal{V} \stackrel{def}{=} \{X/t \in \theta \mid X \in \mathcal{V}\}$ is the *restriction* of θ to $\mathcal{V}$. The convention is that $\circ$ binds stronger than $\restriction$. For instance, $\eta \circ \sigma\!\restriction\! V$ is equal to $(\eta \circ \sigma)\!\restriction\! V$.

An *equation* is a formula of the form $l = r$ where either $l, r \in \mathcal{TERMS}$ or $l, r \in \mathcal{ATOMS}$. The set of all the equations is denoted as $\mathcal{EQ}$. Let $E \in \wp(\mathcal{EQ})$. E is in *solved form* if, for each equation $l = r$ in E, l is a variable that does not occur on the right side of any equation in E. For a set of equations $E \in \wp(\mathcal{EQ})$, $mgu \in \wp(\mathcal{EQ}) \mapsto Sub \cup \{fail\}$ returns either a most general unifier for E if E is unifiable or $fail$ otherwise. $mgu(\{l = r\})$ is sometimes written as $mgu(l, r)$. There is a natural bijection between substitutions and the sets of equations in solved form. $eq(\theta)$ denotes the set of equations in solved form corresponding to a substitution θ. $eq(fail) \stackrel{def}{=} fail$.

A number of collecting semantics have been developed [3, 18, 23, 24, 26, 28] for logic programs. These collecting semantics use different sets of primitive functions such as unification, composition and restriction. Just as those in [6, 15, 28], the collecting semantics for normal logic programs developed in [22] uses only one primitive function which is defined as follows. Let A, B be atoms and θ, ω substitutions.

$$
unify(A,\theta,B,\omega) \stackrel{def}{=}
\left\{\begin{array}{l}
\textit{let } \rho \textit{ be a renaming substitution such that} \\
\quad vars(A\theta\rho) \cap vars(B\omega) = \emptyset \\
\textit{in} \\
\left\{\begin{array}{ll}
\omega \circ mgu(A\theta\rho, B\omega) & \textit{if } mgu(A\theta\rho, B\omega) \neq fail \\
fail & \textit{otherwise}
\end{array}\right.
\end{array}\right.
$$

The collecting semantics associates each textual program point with a set of substitutions. When a program is analysed by abstract interpretation, sets of substitutions are described by abstract substitutions and the primitive function *unify* is safely approximated by another primitive function $unify^\flat$. Since a program analysis is usually, for a textual program point, concerned only with values of variables in the program clause of the program point, it is convenient to parameterise abstract domains with finite sets of variables instead of having a single abstract domain for all abstract substitutions for all textual program points or constructing abstract domains for different textual program points in different ways. Let $< Sub^\flat[\![\mathcal{V}]\!], \sqsubseteq[\![\mathcal{V}]\!] >$ denote the domain for abstract substitutions describing values of variables in $\mathcal{V}$. The idea of using an abstract substitution in $Sub^\flat[\![\mathcal{V}]\!]$ to describe a set of substitutions can be modeled by a monotonic function $\gamma[\![\mathcal{V}]\!]$, called a *concretisation* function, from $Sub^\flat[\![\mathcal{V}]\!]$ to $\wp(Sub)$. A sufficient condition for $unify^\flat$ to safely approximate *unify* is that, for every finite $\mathcal{U}, \mathcal{V} \subseteq \mathcal{VAR}$, every atoms A with $vars(A) \subseteq \mathcal{U}$ and B with $vars(B) \subseteq \mathcal{V}$, every $\theta^\flat \in Sub^\flat[\![\mathcal{U}]\!]$, every $\theta \in \gamma[\![\mathcal{U}]\!](\theta^\flat)$, every $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, and every $\sigma \in \gamma[\![\mathcal{V}]\!](\sigma^\flat)$, if $unify(A,\theta,B,\sigma) \neq fail$ then $unify(A,\theta,B,\sigma) \in \gamma[\![\mathcal{V}]\!](unify^\flat(A,\theta^\flat,B,\sigma^\flat))$.

We shall propose an abstract domain $<Sub^\flat[\![\mathcal{V}]\!], \sqsubseteq[\![\mathcal{V}]\!]>$ for inferring prescriptive types and devise a corresponding primitive function $unify^\flat$. Since the main task of $unify^\flat$ is to approximate unification, we shall call it an abstract unification function. The *procedure-entry* and *procedure-exit* functions in [3], the *type substitution propagation* function in [18, 20] and the corresponding primitive functions in [26] and [28] can be thought of as variants of $unify^\flat$.

3 Abstract Domain

The abstract domain for type, sharing and aliasing information is the direct product of three abstract subdomains for type, sharing and aliasing information respectively. The abstract subdomain for sharing is the pair-sharing domain [17] which also captures non-linearity. The abstract subdomain for aliasing is the set of equivalence relations among variables ordered by the reverse set inclusion $\supseteq$. Section 3.1 introduces the notion of a type and that of a type definition and defines the meaning of a mono-type as a set of terms. Section 3.2 constructs the abstract domain and corresponding concretisation function.

3.1 Types

A *type* is a finite expression denoting a possibly infinite set of values. Values in logic programs are terms. A type denotes a set of terms. Syntactically, a type is a term constructible from a set $\mathcal{CONS}$ of *type constructors* and a denumerable set $\mathcal{PARS}$ of *type parameters*. If a type constructor has an arity of zero then it is called a *base type*. There are two special base types $\check{\top}$ and $\check{\bot}$, denoting $\mathcal{TERMS}$ and $\emptyset$ respectively. A type is a mono-type if it does not contain any type parameter. The set of mono-types is denoted by $\mathcal{MTYPES}$. Mono-types in $\mathcal{MTYPES}$ can be compared according to a partial order that is inductively defined as follows.

Definition 3.1 *Let $M, M_1, \cdots, M_m, M'_1, \cdots, M'_m$ be mono-types and c/m be a type constructor.*

$$
\begin{array}{rcll}
\check{\bot} & \check{\sqsubseteq} & M & \\
M & \check{\sqsubseteq} & \check{\top} & \\
c(M_1,\cdots,M_m) & \check{\sqsubseteq} & c(M'_1,\cdots,M'_m) & \stackrel{def}{=} \\
 & & \forall 1 \leq j \leq m.(M_j \;\check{\sqsubseteq}\; M'_j) &
\end{array}
$$

Example 3.2 *Let $\mathcal{CONS} = \{nat/0, \check{\top}/0, \check{\bot}/0, list/1\}$. Then $\check{\bot} \;\check{\sqsubseteq}\; nat$, $list(nat) \;\check{\sqsubseteq}\; \check{\top}$, $list(\check{\bot}) \;\check{\sqsubseteq}\; list(nat)$ and $nat \;\check{\not\sqsubseteq}\; list(nat)$.*

$<\mathcal{MTYPES}, \check{\sqsubseteq}, \check{\bot}, \check{\top}, \check{\sqcap}, \check{\sqcup}>$ is a complete lattice where, letting $T = c(T_1, \cdots, T_m)$ and $T' = c'(T'_1, \cdots, T'_k)$ be mono-types, the least upper bound operator $\check{\sqcup}$ is

$$
\begin{array}{l}
\check{\bot} \;\check{\sqcup}\; T = T \;\check{\sqcup}\; \check{\bot} = T \\
\check{\top} \;\check{\sqcup}\; T = T \;\check{\sqcup}\; \check{\top} = \check{\top} \\
T \;\check{\sqcup}\; T' = T' \;\check{\sqcup}\; T = \\
\quad \left\{\begin{array}{ll}
c(T_1 \;\check{\sqcup}\; T'_1, \cdots, T_m \;\check{\sqcup}\; T'_m) & \textit{if } c/m = c'/k \\
\check{\top} & \textit{otherwise}
\end{array}\right.
\end{array}
$$

and the greatest lower bound operator $\check{\sqcap}$ is

$$
\begin{array}{lllllll}
\check{\bot} & \check{\sqcap} & T & = & T & \check{\sqcap} & \check{\bot} = \check{\bot} \\
\check{\top} & \check{\sqcap} & T & = & T & \check{\sqcap} & \check{\top} = T \\
T & \check{\sqcap} & T' & = & T' & \check{\sqcap} & T =
\end{array}
$$
$$
\left\{ \begin{array}{ll} c(T_1 \check{\sqcap} T_1', \cdots, T_m \check{\sqcap} T_m') & if\ c/m = c'/k \\ \check{\bot} & otherwise \end{array} \right.
$$

$< \mathcal{MTYPES}, \check{\sqsubseteq} >$ is usually an infinite domain. The computation of the least-fixed point of a function on an infinite domain is beyond of the scope of this paper and the reader is referred to [9].

A type is a general type if it is either a base type except $\check{\bot}$ or of the form $c(\beta_1, \cdots, \beta_m)$ where c/m is a type constructor and $\beta_1, \cdots, \beta_m \in \mathcal{PARS}$ are different type parameters. For instance, $list(\beta)$ is a general type while $list(nat)$ and $list(list(\beta))$ are not. The types of terms and the sets of terms denoted by types are both decided by type definitions for function symbols in Σ.

Definition 3.3 *A type clause is a clause for predicate* $:/2$ *which is of the form*

$$
\begin{array}{r} f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow \\ X_1 : T_1, \cdots, X_n : T_n \end{array}
$$

where $m, n \geq 0$*,* f/n *is a function symbol,* c/m *is a type constructor,* $\beta_1, \cdots, \beta_m$ *are different type parameters,* $X_1, \cdots, X_n$ *are different program variables, and, for* $1 \leq i \leq n$*,* T_i *is either a general type or a type parameter.*

In addition to the above restriction on the form of a type clause, we also require that (i) each function symbol f/n *is typed by exactly one type clause; and (ii)* $vars(T_j) \subseteq \{\beta_1, \cdots, \beta_m\}$ *for* $1 \leq j \leq n$*. (i) disallows overloading [5] and (ii) is often referred to as type preserving [30].*

Example 3.4 *This example illustrates some type definitions.*

$$
\begin{array}{rcll}
0 : nat & \leftarrow & & \%\ D1 \\
s(X) : nat & \leftarrow & X : nat & \%\ D2 \\
[\,] : list(\beta') & \leftarrow & & \%\ D3 \\
[H|T] : list(\beta'') & \leftarrow & H : \beta'', T : list(\beta'') & \%\ D4
\end{array}
$$

D1 says that 0 is a member of type nat. D2 means that if X has type nat then s(X) has type nat. D1 and D2 also gives the set of terms denoted by type nat. D3 states that [ ] is a member of type $list(\beta')$ *for any type* β' *while D4 reads that, for any type* β''*, if H is a member of type* β'' *and T a member of type* $list(\beta'')$ *then* $[H|T]$ *is a member of type* $list(\beta'')$*.*

A type substitution κ is a set of formulae of the form β/T where β is a type parameter and T is a type. If $\kappa = \{\beta_1/T_1, \cdots, \beta_n/T_n\}$ then the domain $dom(\kappa)$ of κ is $\{\beta_1, \cdots, \beta_n\}$. κ is a mono-type substitution if $T_1, \cdots, T_n$ are mono-types. Mono-type substitutions on the same domain are partially ordered by the point-wise extension of the partial order $\check{\sqsubseteq}$ on mono-types. The application of κ to a type is to replace simultaneously each occurrence of each β_i for $1 \leq i \leq n$ in that type with T_i. For instance, $list(\beta_1)\{\beta_1/list(nat), \beta_2/nat\} = list(list(nat))$.

Definition 3.5 *The meaning of a mono-type is defined as follows.*

$$
\check{\gamma}(M) \stackrel{\text{def}}{=} \tag{1}
$$
$$
\left\{ \begin{array}{ll}
\emptyset & if\ M = \check{\bot} \\
\mathcal{TERMS} & if\ M = \check{\top} \\
\left\{ f(t_1, \cdots, t_n) \,\middle|\, \begin{array}{c} f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow \\ X_1 : T_1, \cdots, X_n : T_n \\ \kappa = \{\beta_j/M_j | 1 \leq j \leq m\} \\ \forall 1 \leq i \leq n.(t_i \in \check{\gamma}(T_i\kappa)) \end{array} \right\} & \\
& if\ M = c(M_1, \cdots, M_m)
\end{array} \right.
$$

In the above equation, $\kappa = \{\beta_j/M_j | 1 \leq j \leq m\}$ *is a type substitution. Since* M_j *for* $1 \leq j \leq m$ *is a mono-type and* $\{\beta_1, \cdots, \beta_m\}$ *contains all the type parameters in* T_i *for* $1 \leq i \leq n$*,* $T_i\kappa$ *is a mono-type for* $1 \leq i \leq n$*.*

Example 3.6 *Let* $\mathcal{CONS} = \{nat/0, \check{\top}/0, \check{\bot}/0, list/1\}$*,* $\Sigma = \{0/0, s/1, [\,]/0, ./2\}$*,* $\mathcal{VAR} = \{X, Y, \cdots\}$*, and the type definitions for function symbols in* Σ *be those in example 3.4.*

$$
\begin{array}{rcl}
\check{\gamma}(list(\check{\bot})) & = & \{[\,]\} \\
\check{\gamma}(list(nat)) & = & \{[\,], [0], \cdots, [0, s(0)], \cdots\} \\
\check{\gamma}(list(\check{\top})) & = & \{[\,], \cdots, [X], \cdots, [Y, s(0)], \cdots\}
\end{array}
$$

Lemma 3.7 $\check{\gamma}$ *is monotonic.* ∎

3.2 Abstract Substitutions

A variable typing μ is a set of formulae of the form X/T where X is a variable in $\mathcal{VAR}$ and T is a type. Let $\mu = \{X_1/T_1, \cdots, X_n/T_n\}$. Then $\{X_1, \cdots, X_n\}$ is the domain $dom(\mu)$ of μ. μ is a *variable mono-typing* if T_i is a mono-type for each $1 \leq i \leq n$. Variable mono-typings on the same domain are partially ordered by the point-wise extension of the partial order $\check{\sqsubseteq}$ on mono-types.

Let $\mathcal{V} \subseteq \mathcal{VAR}$ be finite. An abstract substitution $\theta^\flat$ on $\mathcal{V}$ is a triple $<\mathcal{I}, \mathcal{S}, \mathcal{A}>$ where $\mathcal{I}$ is a variable mono-typing on $\mathcal{V}$, $\mathcal{S}$ describes possible sharing among variables in $\mathcal{V}$ and possible non-linearity of variables in $\mathcal{V}$ and $\mathcal{A}$ describes definite aliasing among variables in $\mathcal{V}$.

Let $\mathbf{IS}[\![\mathcal{V}]\!] \stackrel{def}{=} \mathcal{V} \mapsto \mathcal{MTYPES}$ be the set of variable mono-typings on $\mathcal{V}$ and $\gamma_i[\![\mathcal{V}]\!] \in \mathbf{IS}[\![\mathcal{V}]\!] \mapsto \wp(Sub)$ be defined as follows.

$$
\gamma_i[\![\mathcal{V}]\!](\mathcal{I}) \stackrel{def}{=} \{\theta \in Sub \mid \forall X \in \mathcal{V}.(X\theta \in \check{\gamma}(\mathcal{I}(X)))\}
$$

A substitution θ is said to satisfy a variable mono-typing $\mathcal{I}$ on $\mathcal{V}$ if $\theta \in \gamma_i[\![\mathcal{V}]\!](\mathcal{I})$.

Two distinct variables X, Y share under a substitution θ if they are instantiated by θ to two terms that have common variables, i.e., $vars(X\theta) \cap vars(Y\theta) \neq \emptyset$. A variable X is non-linear under a substitution θ if there are repeated occurrences of variables in $X\theta$. Let X be a variable and t a term. $repeat(X, t)$ is true iff X occurs more than once in t. The multiplicity function $\chi \in \mathcal{TERMS} \mapsto \{0, 1, 2\}$ is defined as follows [6].

$$
\chi(t) \stackrel{def}{=} \left\{ \begin{array}{ll} 0 & if\ vars(t) = \emptyset \\ 2 & if\ \exists X \in vars(t).repeat(X, t) \\ 1 & otherwise \end{array} \right.
$$

The possible sharing among variables in $\mathcal{V}$ and possible non-linearity of variables in $\mathcal{V}$ are represented by a symmetric relation $\mathcal{S}$ on $\mathcal{V}$. Such a relation is often called a pair-sharing relation. Let

$$\mathbf{SR}[\![\mathcal{V}]\!] \stackrel{def}{=} \{\mathcal{S} \mid \mathcal{S}\ \textit{is a symmetric relation on}\ \mathcal{V}\}$$

$<\mathbf{SR}[\![\mathcal{V}]\!], \subseteq>$ is a complete lattice. Let $\gamma_s[\![\mathcal{V}]\!] \in \mathbf{SR}[\![\mathcal{V}]\!] \mapsto \wp(Sub)$ be defined as follows.

$$\gamma_s[\![\mathcal{V}]\!](\mathcal{S}) \stackrel{def}{=} \left\{ \theta \in Sub \;\middle|\; \begin{array}{c} \forall X, Y \in \mathcal{V}.(X \not\equiv Y \wedge (X,Y) \notin \mathcal{S} \\ \rightarrow vars(X\theta) \cap vars(Y\theta) = \emptyset) \\ \wedge \\ \forall X \in \mathcal{V}.((X,X) \notin \mathcal{S} \rightarrow \chi(X\theta) \neq 2) \end{array} \right\}$$

A substitution θ is said to satisfy $\mathcal{S}$ if $\theta \in \gamma_s[\![\mathcal{V}]\!](\mathcal{S})$.

Two variables X and Y are aliased under a substitution θ if $X\theta \equiv Y\theta$. The aliasing relationship among variables in $\mathcal{V}$ is an equivalence relation $\mathcal{A}$ on $\mathcal{V}$. Let

$$\mathbf{ER}[\![\mathcal{V}]\!] \stackrel{def}{=} \{\mathcal{A} \mid \mathcal{A}\ \textit{is an equivalence relation on}\ \mathcal{V}\}$$

$<\mathbf{ER}[\![\mathcal{V}]\!], \supseteq>$ is a complete lattice with infimum $\mathcal{V}^2$, supremum $\{<X,X> \mid X \in \mathcal{V}\}$, least upper bound operator $\cap$ and greatest lower bound operator $\lambda \mathcal{A}_1, \mathcal{A}_2 \in \mathbf{ER}[\![\mathcal{V}]\!].(\mathcal{A}_1 \cup \mathcal{A}_2)^*$ where $(\mathcal{A}_1 \cup \mathcal{A}_2)^*$ is the transitive closure of $\mathcal{A}_1 \cup \mathcal{A}_2$. Let $\gamma_a[\![\mathcal{V}]\!] \in \mathbf{ER}[\![\mathcal{V}]\!] \mapsto \wp(Sub)$ be defined as follows.

$$\gamma_a[\![\mathcal{V}]\!](\mathcal{A}) \stackrel{def}{=} \{\theta \in Sub \mid \forall X, Y \in \mathcal{V}.((X,Y) \in \mathcal{A} \rightarrow X\theta \equiv Y\theta)$$

A substitution θ is said to satisfy $\mathcal{A}$ if $\theta \in \gamma_a[\![\mathcal{V}]\!](\mathcal{A})$.

Definition 3.8 *Let $\mathcal{V} \subseteq \mathcal{VAR}$. Define*

$$Sub^\flat[\![\mathcal{V}]\!] \stackrel{\text{def}}{=} \mathbf{IS}[\![\mathcal{V}]\!] \times \mathbf{SR}[\![\mathcal{V}]\!] \times \mathbf{ER}[\![\mathcal{V}]\!]$$

$$\begin{array}{l} <\mathcal{I}_1, \mathcal{S}_1, \mathcal{A}_1> \ \sqsubseteq[\![\mathcal{V}]\!] \ <\mathcal{I}_2, \mathcal{S}_2, \mathcal{A}_2> \stackrel{\text{def}}{=} \\ \quad \forall X \in \mathcal{V}.(\mathcal{I}_1(X) \check{\sqsubseteq} \mathcal{I}_2(X)) \wedge \mathcal{S}_1 \subseteq \mathcal{S}_2 \wedge \mathcal{A}_1 \supseteq \mathcal{A}_2 \end{array}$$

$<Sub^\flat[\![\mathcal{V}]\!], \sqsubseteq[\![\mathcal{V}]\!]>$ is a complete lattice. Let $\theta^\flat = <\mathcal{I}, \mathcal{S}, \mathcal{A}>$. Define $\theta^\flat \downarrow Inst \stackrel{def}{=} \mathcal{I}$, $\theta^\flat \downarrow Share \stackrel{def}{=} \mathcal{S}$ and $\theta^\flat \downarrow Alias \stackrel{def}{=} \mathcal{A}$.

Definition 3.9 *Define*

$$\gamma[\![\mathcal{V}]\!](\theta^\flat) \stackrel{\text{def}}{=} \gamma_i[\![\mathcal{V}]\!](\theta^\flat \downarrow Inst) \cap \gamma_s[\![\mathcal{V}]\!](\theta^\flat \downarrow Share) \cap \gamma_a[\![\mathcal{V}]\!](\theta^\flat \downarrow Alias)$$

$\gamma[\![\mathcal{V}]\!]$ is monotonic. Let $\theta^\flat = <\mathcal{I}, \mathcal{S}, \mathcal{A}>$. A substitution θ is said to satisfy $\theta^\flat$ if $\theta \in \gamma[\![\mathcal{V}]\!](\theta^\flat)$. θ satisfies $\theta^\flat$ iff θ satisfies $\mathcal{I}$, $\mathcal{S}$ and $\mathcal{A}$ simultaneously according to definition 3.9.

We have constructed the abstract domain by the direct product approach [8]. The abstract domain contains abstract substitutions describing the same set of substitutions. In other words, an abstract substitution in the abstract domain may contain redundancy. Following [8], we transform the output abstract substitution of an abstract unification into an equivalent one by reducing redundancy in the output. Another approach to combine abstract domains is the reduced product approach [9] whereby an abstract substitution does not contain redundancy. However, the basic operations used in abstract unifications on component abstract domains have to be redefined in the reduced product approach while they can be directly used in the direct product approach.

4 Preliminary Results and Functions

This section contains some preliminary lemmas and ancillary functions that are used in the abstract unification function.

The lemma 4.1 in the following states that the set of terms described by a mono-type is closed under substitution. In other words, a term will have a smaller mono-type description with respect to $\check{\sqsubseteq}$ as it is getting instantiated during normal unification. This property of mono-types is later used in designing an ancillary function which estimates the effect of normal unification on variable mono-typings.

Lemma 4.1 *$t \in \check{\gamma}(M) \rightarrow t\theta \in \check{\gamma}(M)$ for any $M \in \mathcal{MTYPES}$, any $t \in \mathcal{TERMS}$ and any $\theta \in Sub$.* ∎

The following lemma 4.2 indicates that type information also captures groundness information. Groundness information is used in defining an ancillary function $\chi^\flat$ which approximates the multiplicity function χ. Type information also captures non-freeness since $\check{\top}$ is the most precise mono-type description of a variable. Thus, if a term has a mono-type less than $\check{\top}$ then the term is not a variable.

Lemma 4.2 *Let M be a mono-type. If $\check{\top}$ does not occur in M then t is a ground term for any $t \in \check{\gamma}(M)$.* ∎

Sharing and aliasing together also capture groundness. If two different variables X and Y are aliased together by a substitution σ and X and Y do not share under σ then both X and Y are ground under σ.

Let $<\mathcal{I}, \mathcal{S}, \mathcal{A}>$ be an abstract substitution on $\mathcal{V}$ and $X \in \mathcal{V}$ a variable. Define

$$\mathbf{grd}^t(X, <\mathcal{I}, \mathcal{S}, \mathcal{A}>) \stackrel{def}{=} \left\{ \begin{array}{c} \check{\top}\ \textit{does not occur in}\ \mathcal{I}(X) \\ \vee \\ \exists Y \in \mathcal{V}.(X \not\equiv Y \wedge (X,Y) \in \mathcal{A} \wedge (X,Y) \notin \mathcal{S}) \end{array} \right.$$

By lemma 4.2, for any $\sigma \in \gamma[\![\mathcal{V}]\!](<\mathcal{I}, \mathcal{S}, \mathcal{A}>)$ and any $X \in \mathcal{V}$, if $\mathbf{grd}^t(X, <\mathcal{I}, \mathcal{S}, \mathcal{A}>)$ is true then $X\sigma$ is ground.

We now define the approximate multiplicity function $\chi^\flat$ which, given a term t and an abstract substitution $\sigma^\flat$, estimates $\chi(t\theta)$ for any θ satisfying $\sigma^\flat$. $\chi^\flat$ makes use of groundness information captured by $\sigma^\flat$ and sharing information $\sigma^\flat \downarrow Share$.

Definition 4.3 *Let $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, $X \in \mathcal{V}$ and t' with $vars(t') \subseteq \mathcal{V}$ be a non-variable term. Define*

$$\chi^\flat(X, \sigma^\flat) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} 0 & \textit{if}\ \mathbf{grd}^t(X, \sigma^\flat) \\ 2 & \textit{if}\ (X,X) \in \sigma^\flat \downarrow Share \\ 1 & \textit{otherwise} \end{array} \right.$$

$$\chi^\flat(t', \sigma^\flat) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} 0 & \textit{if}\ \forall X \in vars(t').\chi^\flat(X, \sigma^\flat) = 0 \\ 1 & \textit{if}\ \forall X \in vars(t').(repeat(X, t') \rightarrow \mathbf{grd}^t(X, \sigma^\flat)) \\ & \quad \wedge \\ & \quad \forall Y, Z \in vars(t').(\chi^\flat(Y, \sigma^\flat) = 1 \wedge \\ & \qquad \chi^\flat(Z, \sigma^\flat) = 1 \rightarrow (Y,Z) \notin \sigma^\flat \downarrow Share) \\ 2 & \textit{otherwise} \end{array} \right.$$

By lemma 4.2 and definition 4.3, $\chi^\flat(t,\sigma^\flat) \geq \chi(t\theta)$ for any substitution $\theta \in \gamma[\![\mathcal{V}]\!](\sigma^\flat)$. The definition of $\chi^\flat$ is in essence an improved version of that of Codish et. al [6]. Let all those variables that have multiple occurrences in t are ground under $\sigma^\flat$ and there be no sharing among those variables in t that are not ground under $\sigma^\flat$. Then $\chi^\flat(t,\sigma^\flat) = 1$ according to our definition while $\chi^\flat(t,\sigma^\flat) = 2$ according to the definition of Codish et. al [6].

Aliasing information is used by the abstract unification function to detect failure in normal unification. One rationale for the detection of failure in normal unification is that a term cannot be a proper sub-term of itself. Let t and t' be terms. t' is a proper sub-term of t iff $t' \not\equiv t$ and t' is a sub-term of t. Let $\mathcal{A}$ be an aliasing relation on $\mathcal{V}$, and E be a set of equations of the form $X = t$ where $X \in \mathcal{V}$ is a variable and t is a non-variable term with $vars(t) \subseteq \mathcal{V}$. Define

$$pst(\mathcal{A},E) \stackrel{def}{=} \{<[V]_{\mathcal{A}},[X]_{\mathcal{A}}> \mid (X=t) \in E \wedge V \in vars(t)\}$$

If $<[V]_{\mathcal{A}},[X]_{\mathcal{A}}> \in \ pst(\mathcal{A},E)$ then $V\sigma$ is a proper sub-term of $X\sigma$ for any substitution $\sigma \in \gamma_a[\![\mathcal{V}]\!](\mathcal{A})$. The proper sub-term relationship is transitive. Therefore, if there is a variable X such that $<[X]_{\mathcal{A}},[X]_{\mathcal{A}}> \in \ (pst(\mathcal{A},E))^*$, the transitive closure of $pst(\mathcal{A},E)$, then $E\sigma$ cannot be unified for any $\sigma \in \gamma_a[\![\mathcal{V}]\!](\mathcal{A})$. Another rationale for the detection of failure in normal unification is that if $t_X\sigma \equiv t_Y\sigma$ then t_X and t_Y must unify. Based on the above two rationales, the following function $failure$ makes use of aliasing information to detect failure in normal unification.

$$failure(\mathcal{A},E) \stackrel{def}{=} \left\{ \begin{array}{l} \qquad \exists X. <[X]_{\mathcal{A}},[X]_{\mathcal{A}}> \in (pst(\mathcal{A},E))^* \\ \qquad\qquad \vee \\ \exists (X=t_X),(Y=t_Y) \in E. \\ (X \not\equiv Y \wedge (X,Y) \in \mathcal{A} \wedge mgu(t_X,t_Y) = fail) \end{array} \right.$$

Let $\mathcal{V} \subseteq \mathcal{VAR}$, $\mathcal{A}$ be an equivalence relation and E be a set of equations $X=t$ where $X \in \mathcal{V}$ is a variable and t is a non-variable term such that $vars(t) \subseteq \mathcal{V}$. If $failure(\mathcal{A},E)$ then $mgu(E\sigma) = fail$ for any $\sigma \in \gamma_a[\![\mathcal{V}]\!](\mathcal{A})$.

The definition 4.4 in the following defines an ancillary function $lmts$ which, given a mono-type M and a general type or type parameter G, gives the least (with respect to the point-wise extension of $\check{\sqsubseteq}$) mono-type substitution κ such that $M \check{\sqsubseteq} G\kappa$ if one exists.

Definition 4.4 *Let M be a mono-type and G a general type or a type parameter. Define*

$$lmts(M,G) \stackrel{\rm def}{=} \left\{ \begin{array}{ll} \emptyset & if\ G = \check{\top} \\ \{\beta/\check{\bot} \mid \beta \in vars(G)\} & if\ M = \check{\bot} \\ \{G/M\} & if\ G \in \mathcal{PARS} \\ \bigcup_{1\leq j\leq m}\{\beta_j/M_j\} & if\ \ M = c(M_1,\cdots,M_m) \\ & \quad \wedge\ G = c(\beta_1,\cdots,\beta_m) \\ fail & otherwise \end{array} \right.$$

Example 4.5 *Let $\beta,\beta_1,\beta_2 \in \mathcal{PARS}$ and $\mathcal{CONS} = \{nat/0, \check{\top}/0, \check{\bot}/0, list/1, pair/2\}$.*

$$\begin{array}{l} lmts(pair(nat,list(\check{\top})),pair(\beta_1,\beta_2)) = \\ \qquad\qquad \{\beta_1/nat, \beta_2/list(\check{\top})\} \\ lmts(list(\check{\top}),pair(\beta_1,\beta_2)) = fail \end{array}$$

Lemma 4.6 below follows from definitions 3.1 and 4.4. (a) and (d) in lemma 4.6 states that if $lmts(M,G) \neq fail$ then it is the least mono-type substitution under which the instance of G is greater than or equal to M. (b) and (c) in lemma 4.6 implies that $lmts$ is monotonic on its first argument.

Lemma 4.6 *If M, M' are mono-types, G is a general type or a type parameter, and κ,κ',κ'' are mono-type substitutions such that*

$$M \check{\sqsubseteq} M', \quad \kappa = lmts(M,G), \quad \kappa' = lmts(M',G)$$

then

(a) $\kappa \neq fail \rightarrow M \check{\sqsubseteq} G\kappa$.

(b) $\kappa = fail \rightarrow \kappa' = fail$.

(c) $\kappa \neq fail \rightarrow (\kappa' = fail \vee \forall \beta \in vars(G).(\beta\kappa \check{\sqsubseteq} \beta\kappa'))$.

(d) $M \check{\sqsubseteq} G\kappa'' \rightarrow (\kappa \neq fail \wedge \forall \beta \in vars(G).(\beta\kappa \check{\sqsubseteq} \beta\kappa''))$.

∎

The abstract unification function given in section 5 needs an ancillary function which, given mono-type descriptions of the variables in a term, finds a mono-type description of the term. The function $\check{A}$ given in definition 4.7 is such a function. Definition 4.7 also gives a function $\check{\alpha}$ which finds a mono-type description of a term without knowing mono-type descriptions of the variables in the term.

Definition 4.7 *Let t be a term and μ a variable mono-typing such that $vars(t) \subseteq dom(\mu)$. Define*

$$\check{\alpha}(t) \stackrel{\rm def}{=} \check{A}(t, \{X/\check{\top} \mid X \in vars(t)\}) \tag{2}$$

$$\check{A}(t,\mu) \stackrel{\rm def}{=} \left\{ \begin{array}{ll} \mu(t) & if\ t\ is\ a\ variable \\ c(\beta_1,\cdots,\beta_m)\kappa & \\ \quad if\ t = f(t_1,\cdots,t_n) & \\ \quad \wedge\ f(X_1,\cdots,X_n) : c(\beta_1,\cdots,\beta_m) \leftarrow & \\ \qquad\qquad X_1 : T_1,\cdots,X_n : T_n & \\ \quad \wedge\ \kappa_i = lmts(\check{A}(t_i,\mu),T_i) & \\ \quad \wedge\ \forall 1 \leq i \leq n.\kappa_i \neq fail & \\ \quad \wedge\ \kappa = \bigcup_{1\leq j\leq m}\{\beta_j/(\check{\bigsqcup}_{\substack{1\leq i\leq n \\ \beta_j \in dom(\kappa_i)}}\{\beta_j\kappa_i\})\} & \\ \check{\top} & otherwise \end{array} \right. \tag{3}$$

For any $\theta \in Sub$, $t\theta$ is of type $\check{A}(t,\mu)$ if, for each variable $X \in vars(t)$, $X\theta$ is of type $\mu(X)$. For any $\theta \in Sub$, $t\theta$ is of type $\check{\alpha}(t)$.

Example 4.8 *Let the type definitions be those in example 3.4. The computation of $\check{\alpha}([0,X])$ involves the following steps.*

(1) $\check{A}(0,\{X/\check{\top}\}) = nat$;

(2) $lmts(nat,\beta'') = \{\beta''/nat\}$;

(3) compute $\check{A}([X],\{X/\check{\top}\})$ *as follows;*

(i) $\check{A}(X,\{X/\check{\top}\}) = \check{\top}$ *since X is a variable;*

(ii) $lmts(\check{\top},\beta'') = \{\beta''/\check{\top}\}$;

(iii) $\check{A}([\,],\{X/\check{\top}\}) = list(\beta')\{\beta'/\check{\bigsqcup}\emptyset\}$
$= list(\beta')\{\beta'/\check{\bot}\} = list(\check{\bot})$;

(iv) $lmts(list(\check{\bot}), list(\beta'')) = \{\beta''/\check{\bot}\}$;

(v) $\check{A}([X],\{X/\check{\top}\}) = list(\beta'')\{\beta''/(\check{\top}\ \check{\sqcup}\ \check{\bot})\}$
$= list(\check{\top})$.

(4) $lmts(list(\check{\top}), list(\beta'')) = \{\beta''/\check{\top}\}$;

(5) $\check{\alpha}([0,X]) = \check{A}([0,X],\{X/\check{\top}\})$
$= list(\beta'')\{\beta''/(nat\ \check{\sqcup}\ \check{\top})\} = list(\check{\top})$.

The lemmas 4.9-4.11 given below summaries some properties of $\check{A}$ and $\check{\alpha}$. Lemma 4.9 states that $\check{A}$ is monotonic on its second argument. Lemma 4.10.(a) ascertains the soundness of $\check{A}$ and $\check{\alpha}$ and lemma 4.10.(b) restates that the set of terms described by a mono-type is closed under substitution. It is implied by lemma 4.11 that there is a Galois connection between the set of sets of terms ordered by set inclusion $\subseteq$ and the set of mono-types ordered by $\check{\sqsubseteq}$.

Lemma 4.9 *Let t be a term and μ,ν be variable mono-typings. If $vars(t) \subseteq dom(\mu)$, $vars(t) \subseteq dom(\nu)$ and $\forall X \in vars(t).(\mu(X) \check{\sqsubseteq} \nu(X))$ then $\check{A}(t,\mu) \check{\sqsubseteq} \check{A}(t,\nu)$.* ∎

Lemma 4.10 *For any term t, any substitution θ and any variable mono-typing μ,*

(a) $vars(t) \subseteq dom(\mu) \wedge \forall X \in vars(t).(\check{\alpha}(X\theta) \check{\sqsubseteq} \mu(X)) \rightarrow \check{\alpha}(t\theta) \check{\sqsubseteq} \check{A}(t,\mu)$.

(b) $\check{\alpha}(t\theta) \check{\sqsubseteq} \check{\alpha}(t)$.

Proof: By equations 2 and 3,

$$\begin{aligned}\check{\alpha}(t) &= \check{A}(t,\{X/\check{\top} \mid X \in vars(t)\})\\ \check{\alpha}(t\theta) &= \check{A}(t,\{X/\check{\alpha}(X\theta) \mid X \in vars(t)\})\end{aligned}$$

Therefore, (a) and (b) hold by lemma 4.9. ∎

Lemma 4.11 *For any term t and any mono-type M, $t \in \check{\gamma}(M) \leftrightarrow \check{\alpha}(t) \check{\sqsubseteq} M$.* ∎

Given a term and a mono-type, the function *gvmt* defined below computes the greatest variable mono-typing on the set of the variables in the term such that the term is described by the mono-type.

Definition 4.12 *Let t be a term and M be a mono-type. Define*

$$gvmt(M,t) \stackrel{\text{def}}{=} \tag{4}$$
$$\begin{cases} \{X/M \mid X \in vars(t)\} & if\ M = \check{\top} \vee t \in \mathcal{VAR}\\ \bigcup_{X\in vars(t)}\{X/(\check{\sqcap}_{\substack{1\le i\le n\\ X\in dom(\mu_i)}}\{\mu_i(X)\})\} & \\ \quad if\ \ t = f(t_1,\cdots,t_n) & \\ \quad \wedge\ f(X_1,\cdots,X_n) : c(\beta_1,\cdots,\beta_m) \leftarrow & \\ \qquad\qquad X_1 : T_1,\cdots,X_n : T_n & \\ \quad \wedge\ M = c(M_1,\cdots,M_m) & \\ \quad \wedge\ \kappa = \{\beta_j/M_j \mid 1 \le j \le m\} & \\ \quad \wedge\ \forall 1 \le i \le n.(\mu_i = gvmt(T_i\kappa, t_i)) & \\ \{X/\check{\bot} \mid X \in vars(t)\} & otherwise \end{cases}$$

Example 4.13 *Let the type definitions be those in example 3.4. The computation of $gvmt(list(nat),[X,Y|X])$ involves the following steps and makes use of the type definitions D3 and D4.*

(1) $gvmt(nat,X) = \{X/nat\}$;

(2) compute $gvmt(list(nat),[Y|X])$;

(i) $gvmt(nat,Y) = \{Y/nat\}$;

(ii) $gvmt(list(nat),X) = \{X/list(nat)\}$;

(iii) $gvmt(list(nat),[Y|X]) = \{X/list(nat), Y/nat\}$.

(3) $gvmt(list(nat),[X,Y|X]) = \{X/\check{\bot}, Y/nat\}$.

For any $\theta \in Sub$ and any variable $X \in vars(t)$, $X\theta$ must be of type $gvmt(M,t)(X)$ in order for $t\theta$ to be of type M. This is stated in the following by lemma 4.14.

Lemma 4.14 *Let t be a term, θ a substitution, M a mono-type and $\nu = gvmt(M,t)$.*

$$\check{\alpha}(t\theta) \check{\sqsubseteq} M \rightarrow \forall X \in vars(t).(\check{\alpha}(X\theta) \check{\sqsubseteq} \nu(X))$$

∎

We conclude this section by defining and proving correct a function $\mathbf{est}^t$ which estimates the effect of the unification of an equation in solved form on the mono-type descriptions of the variables in the equation. Given an equation in solved form and mono-type descriptions of the variables in the equation before unification, $\mathbf{est}^t$ computes mono-type descriptions of the variables in the equation after unification.

Definition 4.15 *Let μ be a variable mono-typing on $\mathcal{V}$, $X \in \mathcal{V}$ and $vars(t) \subseteq \mathcal{V}$*

$$\mathbf{est}^t(X = t,\mu) \stackrel{\text{def}}{=} \tag{5}$$
$$\begin{cases} let\ \mathcal{V}_t = vars(t),\\ \quad \mu_1 = gvmt(\mu(X),t),\\ \quad \mu_2 = \{Y/(\mu(Y)\check{\sqcap}\mu_1(Y)) \mid Y \in \mathcal{V}_t\}\\ in\\ \lambda Y.\begin{cases}\mu(Y)\check{\sqcap}\check{A}(t,\mu_2) & if\ Y \equiv X\\ \mu_2(Y) & if\ Y \in \mathcal{V}_t\end{cases}\end{cases}$$

Lemma 4.16 *If $X \in \mathcal{V}$, $vars(t) \subseteq \mathcal{V}$, $\mathcal{I} \in \mathbf{IS}[\![\mathcal{V}]\!]$, $\sigma \in \gamma_i[\![\mathcal{V}]\!](\mathcal{I})$, $\eta = mgu(X\sigma, t\sigma) \neq fail$ and $\mathcal{I}_0 = \mathbf{est}^t(X = t,\mathcal{I})$ then $\forall Y \in vars(t) \cup \{X\}.Y\sigma\eta \in \check{\gamma}(\mathcal{I}_0(Y))$.* ∎

5 Abstract Unification Function

Let $\mathcal{U},\mathcal{V} \subseteq \mathcal{VAR}$, $\theta^\flat \in Sub^\flat[\![\mathcal{U}]\!]$, $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, and A, B be atoms such that $vars(A) \subseteq \mathcal{U}$ and $vars(B) \subseteq \mathcal{V}$. The abstract unification function $unify^\flat$ computes $unify^\flat(A,\theta^\flat,B,\sigma^\flat)$ in three stages. Let Ψ be a renaming substitution such that $\mathcal{U}\Psi \cap \mathcal{V} = \emptyset$. In the first stage, it computes $E_0 = eq(mgu(\{A\Psi = B\}))$. If $E_0 = fail$ then it returns $\bot[\![\mathcal{V}]\!]$ since $\bot[\![\mathcal{V}]\!]$ approximates the empty set $\emptyset$ of substitutions more precisely than any other abstract substitution in $Sub^\flat[\![\mathcal{V}]\!]$. In the second stage, letting $\mathcal{W} = \mathcal{U}\Psi \cup \mathcal{V}$ and $\theta^\flat\Psi \uplus \sigma^\flat \in Sub^\flat[\![\mathcal{W}]\!]$ be the component-wise unions of $\theta^\flat\Psi$ and $\sigma^\flat$, it computes an abstract substitution $\eta^\flat \in Sub^\flat[\![\mathcal{W}]\!]$ such that, $\zeta \circ mgu(E_0\zeta) \in \gamma[\![\mathcal{W}]\!](\eta^\flat)$ for any $\zeta \in \gamma[\![\mathcal{W}]\!](\theta^\flat \uplus \sigma^\flat)$. This is accomplished by a function *solve*. In the third stage, it restricts $\eta^\flat$ to $\mathcal{V}$. This is accomplished by a function *restrict*. Note that $vars(\sigma^\flat \downarrow Alias) = \mathcal{V}$ for any $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$. The following is the definition of the abstract unification function $unify^\flat$.

Let $\mathcal{U}, \mathcal{V} \subseteq \mathcal{VAR}$, $\theta^\flat \in Sub^\flat[\![\mathcal{U}]\!]$, $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, and A, B be atoms such that $vars(A) \subseteq \mathcal{U}$ and $vars(B) \subseteq \mathcal{V}$.

$$unify^\flat(A, \theta^\flat, B, \sigma^\flat) \stackrel{def}{=} \tag{6}$$
$$\left\{ \begin{array}{l} let \quad \Psi \ be \ a \ renaming \ substitution \ such \ that \\ \qquad \mathcal{U}\Psi \cap \mathcal{V} = \emptyset, \\ \qquad E_0 = eq(mgu(A\Psi, B)), \\ \qquad \mathcal{V} = vars(\sigma^\flat \downarrow Alias) \\ in \\ \left\{ \begin{array}{ll} restrict(solve(E_0, \theta^\flat\Psi \uplus \sigma^\flat), \mathcal{V}) & if \ E_0 \neq fail \\ \bot[\![\mathcal{V}]\!] & otherwise \end{array} \right. \end{array} \right.$$

$$<\mathcal{I}, \mathcal{S}, \mathcal{A}> \uplus <\mathcal{I}', \mathcal{S}', \mathcal{A}'> \stackrel{def}{=} <\mathcal{I} \cup \mathcal{I}', \mathcal{S} \cup \mathcal{S}', \mathcal{A} \cup \mathcal{A}'> \tag{7}$$

$$solve(E, \theta^\flat) \stackrel{def}{=} \tag{8}$$
$$\left\{ \begin{array}{l} let \quad E_1 = \{(X = Y) \in E \mid Y \ \in \mathcal{VAR}\}, \\ \qquad E_2 = E \setminus E_1, \\ \qquad \zeta^\flat = solve0(E_1, \theta^\flat) \\ in \\ \left\{ \begin{array}{ll} \bot[\![vars(\theta^\flat \downarrow Alias)]\!] & if \ failure(\zeta^\flat \downarrow Alias, E_2) \\ solve0(E_2, \zeta^\flat) & otherwise \end{array} \right. \end{array} \right.$$

$$solve0(E, \theta^\flat) \stackrel{def}{=} \tag{9}$$
$$\left\{ \begin{array}{ll} \theta^\flat & if \ E = \emptyset \\ solve0(E', soln(e, \theta^\flat)) & if \ E = \{e\} \cup E' \end{array} \right.$$

$$restrict(<\mathcal{I}, \mathcal{S}, \mathcal{A}>, \mathcal{V}) \stackrel{def}{=} \tag{10}$$
$$<\{X/\mathcal{I}(X) \mid X \in \mathcal{V}\}, \mathcal{S} \cap \mathcal{V}^2, \mathcal{A} \cap \mathcal{V}^2>$$

$$soln(e, \sigma^\flat) \stackrel{def}{=} reduce(compose(\sigma^\flat, unieq(e, \sigma^\flat))) \tag{11}$$

$$unieq(X = t, <\mathcal{I}, \mathcal{S}, \mathcal{A}>) \stackrel{def}{=} <\mathcal{I}', \mathcal{S}', \mathcal{A}'> \tag{12}$$

where

$$\mathcal{I}' = \left\{ \begin{array}{l} let \ \mathcal{I}_0 = \mathbf{est}^t(X = t, \mathcal{I}) \\ \quad \mathcal{Y} = vars(t) \cup \{X\} \\ in \\ \lambda U. \left\{ \begin{array}{ll} \mathcal{I}_0(Y) & if \ \exists Y \in \mathcal{Y}.((U, Y) \in \mathcal{A}) \\ \mathcal{I}(U) & otherwise \end{array} \right. \end{array} \right.$$

$$\mathcal{S}' = \left\{ \begin{array}{l} let \ \mathcal{V}_t = vars(t), \\ \quad \kappa^1 = \chi^\flat(X, <\mathcal{I}, \mathcal{S}, \mathcal{A}>), \\ \quad \kappa^2 = \chi^\flat(t, <\mathcal{I}, \mathcal{S}, \mathcal{A}>), \\ \quad \kappa = (\kappa^1, \kappa^2) \\ in \\ \left\{ \begin{array}{ll} \{(X, Y), (Y, X) | Y \in \mathcal{V}_t\} & if \ \kappa = (1, 1) \\ \{(X, Y), (Y, X) | Y \in \mathcal{V}_t\} \cup \{(X, X)\} & \\ & if \ \kappa = (1, 2) \\ \{(X, Y), (Y, X) | Y \in \mathcal{V}_t\} \cup \mathcal{V}_t^2 & if \ \kappa = (2, 1) \\ \{(X, Y), (Y, X) | Y \in \mathcal{V}_t\} \cup \{(X, X)\} \cup \mathcal{V}_t^2 & \\ & if \ \kappa = (2, 2) \\ \emptyset & otherwise \end{array} \right. \end{array} \right.$$

$$\mathcal{A}' = \left\{ \begin{array}{ll} (\{(X, t), (t, X)\} \cup \mathcal{A})^* & if \ t \in \mathcal{VAR} \\ \mathcal{A} & otherwise \end{array} \right.$$

$$compose(<\mathcal{I}_i, \mathcal{S}_i, \mathcal{A}_i>, <\mathcal{I}', \mathcal{S}', \mathcal{A}'>) \stackrel{def}{=} \tag{13}$$
$$\begin{array}{l} <\mathcal{I}', \\ \left[\begin{array}{c} \mathcal{S}_i \ \cup \ \mathcal{S}' \cup \\ \{(U, Z) | \exists V.((U, V) \in \mathcal{S}' \wedge (V, Z) \in \mathcal{S}_i)\} \cup \\ \{(W, V) | \exists U.((W, U) \in \mathcal{S}_i \wedge (U, V) \in \mathcal{S}')\} \cup \\ \left\{ (W, Z) \left| \begin{array}{l} \exists U, V.((W, U) \in \mathcal{S}_i \wedge \\ (U, V) \in \mathcal{S}' \wedge (V, Z) \in \mathcal{S}_i) \end{array} \right. \right\} \end{array} \right], \\ \mathcal{A}'> \end{array}$$

$$reduce(<\mathcal{I}, \mathcal{S}, \mathcal{A}>) \stackrel{def}{=} \tag{14}$$
$$\left\{ \begin{array}{ll} \bot[\![vars(\mathcal{A})]\!] & if \ \exists Y.\mathcal{I}(Y) = \check{\bot} \\ <\mathcal{I}, \left\{ (X, Y) \in \mathcal{S} \left| \begin{array}{c} \neg\mathbf{grd}^t(X, <\mathcal{I}, \mathcal{S}, \mathcal{A}>) \\ \wedge \\ \neg\mathbf{grd}^t(Y, <\mathcal{I}, \mathcal{S}, \mathcal{A}>) \end{array} \right. \right\}, \mathcal{A}> & \\ & otherwise \end{array} \right.$$

solve can be thought of as an approximate constraint solver [6, 7]. A constraint is a pair $(E, \theta^\flat)$ where E is a set of equations in solved form and $\theta^\flat$ is an abstract substitution. A substitution is said to satisfy $(E, \theta^\flat)$ iff it is equivalent (modulo renaming) to $\theta \circ mgu(E\theta)$ for some $\theta \in \gamma[\![\mathcal{W}]\!](\theta^\flat)$. What *solve* does is to transform $(E, \theta^\flat)$ into $(\emptyset, \eta^\flat)$ such that any substitution satisfying $(E, \theta^\flat)$ also satisfies $(\emptyset, \eta^\flat)$. Note that the set of substitutions satisfying $(\emptyset, \eta^\flat)$ is $\gamma[\![\mathcal{W}]\!](\eta^\flat)$. *solve* does so by first calling *solve*0 to transform $(E, \theta^\flat)$ to $(E_2, \zeta^\flat)$ such that any substitution satisfying $(E, \theta^\flat)$ satisfies $(E_2, \zeta^\flat)$ where E_2 be the set of the equations in E whose right sides are not variables, then calling *failure* to detect failure in normal unification according to information in E_2 and $\zeta^\flat \downarrow Alias$, and then calling *solve*0 to transform $(E_2, \zeta^\flat)$ to $(\emptyset, \eta^\flat)$ such that any substitution satisfying $(E_2, \zeta^\flat)$ satisfies $(\emptyset, \eta^\flat)$. *solve*0 repeatedly calls *soln* to transform a constraint of the form $(E \cup \{e\}, \theta^\flat)$ into a constraint of the form $(E, \sigma^\flat)$ such that any substitution satisfying $(E \cup \{e\}, \theta^\flat)$ satisfies $(E, \sigma^\flat)$.

Let $\sigma^\flat \in Sub^\flat[\![\mathcal{W}]\!]$ and $vars(e) \subseteq \mathcal{W}$. The rationale for $soln(e, \sigma^\flat)$ is to ensure that, for any $\sigma \in \gamma[\![\mathcal{W}]\!](\sigma^\flat)$, if $\eta = mgu(\{e\sigma\}) \neq fail$ then $\sigma \circ \eta \in \gamma[\![\mathcal{W}]\!](soln(e, \sigma^\flat))$. The unification of $\{e\sigma\}$ will possibly

(i) change the types of the values of the variables in $vars(e)$,

(ii) cause more pair-sharing among the variables in $vars(e)$, and

(iii) cause more aliasing among the variables in $vars(e)$.

The composition of $\eta = mgu(\{e\sigma\})$ to σ will possibly

(iv) change the types of the values of the variables in $\mathcal{W} \setminus vars(e)$,

(v) cause more pair-sharing among the variables in $\mathcal{W} \setminus vars(e)$ and/or more pair-sharing between the variables in $vars(e)$ and the variables in $\mathcal{W} \setminus vars(e)$, and

(vi) cause more aliasing among the variables in $\mathcal{W} \setminus vars(e)$ and/or more aliasing between the variables in $vars(e)$ and the variables in $\mathcal{W} \setminus vars(e)$.

The effects of (i), (ii), (iii) and (vi) are estimated by a function *unieq*. *unieq* also estimates part of the effect of (iv) - estimating the types of the values of those variables in $\mathcal{W} \setminus vars(e)$ which are aliased with variables in $vars(e)$ by σ. That means that if the two input abstract substitutions describes each pair of aliased variables with the same mono-type then each pair of aliased variables have the same mono-type description after each time *soln* is called. The effect of (v) is estimated by a function *compose*. Finally, *soln* calls *reduce* to transform its output abstract substitution into an equivalent one with less redundancy. Let $\sigma^\flat = <\mathcal{I}, \mathcal{S}, \mathcal{A}>$ be an abstract substitution. If $\mathcal{I}(Y) = \check{\perp}$ for some Y then *reduce* transforms $\sigma^\flat$ into $\perp[\![vars(\mathcal{A})]\!]$ since $\sigma^\flat$ describes the empty set of substitution. Otherwise, *reduce* removes a sharing pair (X, Y) from $\mathcal{S}$ if either X or Y is ground under $\sigma^\flat$.

The presence of aliasing information allows *unieq* to make more accurate estimation of the effect of (vi). Let $Y \in \mathcal{W} \setminus vars(e)$. By lemma 4.1, any type of $Y\sigma$ is also a type of $Y\sigma\eta$. Therefore, the effect of (vi) can be ignored as in [14, 19] as far as the safety of type analysis is concerned. If Y is aliased with a variable in $vars(e)$, *unieq* may make a more accurate estimation of the type of $Y\sigma\eta$. If Y is not aliased with any variable in $vars(e)$, *unieq* takes the type of $Y\sigma$ as the type of $Y\sigma\eta$.

Example 5.1 *This example shows the computation of* $unify^\flat(A, \theta^\flat, B, \sigma^\flat)$. *Let the type definitions be those in example 3.4 and*

$$\begin{array}{rcl}
\mathcal{U} & = & \{W, Y, Z\} \\
\mathcal{V} & = & \{X, W, Y, Z\} \\
A & = & append(W, Y, Z) \\
B & = & append([X|W], Y, [X|Z]) \\
\theta^\flat & = & \left\langle \begin{array}{c} \{W/list(nat), Y/list(nat), Z/\check{\top}\}, \\ \{(Z, Z)\}, \\ \{(W, W), (Y, Y), (Z, Z)\} \end{array} \right\rangle \\
\sigma^\flat & = & \left\langle \begin{array}{c} \{W/\check{\top}, X/\check{\top}, Y/\check{\top}, Z/\check{\top}\}, \emptyset, \\ \{(W, W), \\ (X, X), (Y, Y), (Z, Z)\} \end{array} \right\rangle
\end{array}$$

In stage (1), $\Psi = \{W/W_0, Y/Y_0, Z/Z_0\}$ *is applied to* A *and* $\theta^\flat$.

$$\begin{array}{rcl}
A\Psi & = & append(W_0, Y_0, Z_0) \\
\theta^\flat\Psi & = & \left\langle \begin{array}{c} \{W_0/list(nat), Y_0/list(nat), Z_0/\check{\top}\}, \\ \{(Z_0, Z_0)\}, \\ \{(W_0, W_0), (Y_0, Y_0), (Z_0, Z_0)\} \end{array} \right\rangle \\
E_0 & = & eq(mgu(A\Psi, B)) \\
& = & \{W_0 = [X|W], Y_0 = Y, Z_0 = [X|Z]\}
\end{array}$$

In stage (2),

$$\begin{array}{rcl}
\zeta^\flat & = & \theta^\flat\Psi \uplus \sigma^\flat \\
& = & \left\langle \begin{array}{c} \left\{ \begin{array}{c} W_0/list(nat), Y_0/list(nat), \\ Z_0/\check{\top}, W/\check{\top}, X/\check{\top}, Y/\check{\top}, Z/\check{\top} \end{array} \right\}, \\ \{(Z_0, Z_0)\}, \\ \left\{ \begin{array}{c} (W_0, W_0), (Y_0, Y_0), (Z_0, Z_0), \\ (W, W), (X, X), (Y, Y), (Z, Z) \end{array} \right\} \end{array} \right\rangle
\end{array}$$

and, $\eta^\flat = solve(E_0, \zeta^\flat)$ *is computed. This is accomplished by calling soln via solve0 to solve one equation in* E_0 *after another with* $\zeta^\flat$ *as the constraint for the first equation and the intermediate results as the constraints for other equations.*

$$\eta^\flat = \left\langle \begin{array}{c} \left\{ \begin{array}{c} W_0/list(nat), Y_0/list(nat), Z_0/\check{\top}, \\ W/list(nat), X/nat, Y/list(nat), Z/\check{\top} \end{array} \right\}, \\ \{(Z_0, Z_0), (Z_0, Z), (Z, Z_0), (Z, Z)\}, \\ \left\{ \begin{array}{c} (W_0, W_0), (Y_0, Y_0), (Z_0, Z_0), (W, W), \\ (X, X), (Y, Y), (Z, Z), (Y_0, Y), (Y, Y_0) \end{array} \right\} \end{array} \right\rangle$$

In stage (3),

$$unify^\flat(A, \theta^\flat, B, \sigma^\flat) = restrict(\eta^\flat, \mathcal{V}) = \left\langle \begin{array}{c} \{W/list(nat), X/nat, Y/list(nat), Z/\check{\top}\}, \\ \{(Z, Z)\} \\ \{(W, W), (X, X), (Y, Y), (Z, Z)\} \end{array} \right\rangle$$

The following theorem states that $unify^\flat$ is a safe approximation of *unify*.

Theorem 5.2 *Let* $\mathcal{U}, \mathcal{V} \subseteq \mathcal{VAR}$ *be finite,* A *and* B *be atoms with* $vars(A) \subseteq \mathcal{U}$ *and* $vars(B) \subseteq \mathcal{V}$, $\theta^\flat \in Sub^\flat[\![\mathcal{U}]\!]$, $\theta \in \gamma[\![\mathcal{U}]\!](\theta^\flat)$, $\sigma^\flat \in Sub^\flat[\![\mathcal{V}]\!]$, $\sigma \in \gamma[\![\mathcal{V}]\!](\sigma^\flat)$. *If* $unify(A, \theta, B, \sigma) \neq fail$ *then* $unify(A, \theta, B, \sigma) \in \gamma[\![\mathcal{V}]\!](unify^\flat(A, \theta^\flat, B, \sigma^\flat))$. ∎

6 Conclusion

We have developed an abstract domain and an abstract unification function for inferring type information together with sharing and aliasing information. The idea to incorporate type definitions in type analysis originates from the work by Horiuchi and Kanamori [14, 19]. However, there are three main differences between our work and their work. Firstly, an abstract substitution in their type analysis captures only type information while an abstract substitution in our type analysis captures sharing and aliasing information as well as type information. Secondly, we use aliasing information to detect failure in normal unification and to make more accurate estimation of the effect of unification on the types of values of variables. Thirdly, given a constraint which is a pair consisting of a set of equations in solved form and an abstract substitution, their abstract unification function solves it in one step while our abstract unification function solves the constraint step by step by removing one equation in one step. That is, our abstract unification function procedurally resembles the normal unification function and can be thought of as an abstraction of the normal unification function.

References

[1] H. Azzoune. Type inference in Prolog. In E. Lusk and R. Overbeek, editors, *Proceedings of the ninth International Conference on Automated Deduction*, pages 258–277, Argonne, Illinois, USA, May 23-26 1988. Springer-Verlag. Springer-Verlag Lecture Notes in Computer Science 310.

[2] M. Bruynooghe. Adding redundancy to obtain more reliable and more readable Prolog programs. In *Proceedings of the first International Logic Programming Conference*, pages 129–133, Marseille, France, 1982.

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

[4] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: towards the global optimisation of Prolog programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204. The IEEE Society Press, 1987.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM computing surveys*, 17(4):471–522, 1985.

[6] M. Codish, D. Dams, and Yardeni E. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 79–93, Paris, France, 1991. The MIT Press.

[7] M. Codish, D. Dams, G. Filé, and M. Bruynooghe. Freeness analysis for logic programs - and correctness. In D.S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 117–131. The MIT Press, 1993.

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

[9] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.

[10] P.W. Dart and J. Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. MIT Press, Cambridge,Massachusetts, 1992. ISBN 0-262-16131-1.

[11] T.W. Früehwirth. Using meta-interpreters for polymorphic type checking. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-programming in Logic*, pages 339–351, Leuven, Belgium, April 4-6 1990.

[12] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *The seventh Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 17-19 1990. The ACM Press.

[13] N. Heintze and J. Jaffar. Semantic types for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. MIT Press, Cambridge,Massachusetts, 1992. ISBN 0-262-16131-1.

[14] K. Horiuchi and T. Kanamori. Polymorphic type inference in Prolog by abstract interpretation. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Proceedings of the Sixth Conference on Logic Programming*, pages 195–214, Tokyo, June 1987.

[15] D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *Journal of Logic Programming*, 13(1, 2, 3 and 4):291–314, 1992.

[16] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(1, 2, 3 and 4):205–258, 1992.

[17] N.D. Jones and H. Søndergarrd. A semantics-based framework for abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 123–142. Ellis Horwood Limited, 1987.

[18] T. Kanamori. Abstract interpretation based on Alexander templates. *Journal of Logic Programming*, 15(1 & 2):31–54, January 1993.

[19] T. Kanamori and K. Horiuchi. Type inference in Prolog and its application. In *Proceedings of the ninth International Joint Conference on Artificial Intelligence*, pages 704–707, 1985.

[20] T. Kanamori and T. Kawamura. Abstract interpretation based on oldt resolution. *Journal of Logic Programming*, 15(1 & 2):1–30, January 1993.

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

[22] L. Lu. Abstract interpretation, bug detection and bug diagnosis in normal logic programs. PhD thesis, University of Birmingham, 1994.

[23] K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of normal logic programs. *Journal of Logic Programming*, 13(1, 2, 3 and 4):181–204, 1992.

[24] C. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 181–198. Ellis Horwood Limited, 1987.

[25] P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE international Symposium on Logic Programming*, pages 289–298. IEEE, 1984.

[26] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(1, 2, 3 and 4):315–347, 1992.

[27] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[28] U. Nilsson. Towards a framework for the abstract interpretation of loic programs. In P. Deransart, B. Lorho, and J. Małuszynski, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, pages 68–82, Orléans, France, 1988. Springer-Verlag.

[29] T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34(1):227–240, 1984.

[30] J. Tiuryn. Type inference problems: A survey. In B. Roven, editor, *Proceedings of the Fifteenth International Symposium on Mathematical Foundations of Computer Science*, pages 105–120, Berlin,Heidelberg, 1990. Springer-Verlag.

[31] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Artificial Intelligence*, 23(10):733–742, 1976.

[32] J. Xu and D.S. Warren. A type inference system for Prolog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the fifth International Conference and Symposium on Logic Programming*, pages 604–619. The MIT Press, 1988.

[33] E. Yardeni and E. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–153, 1991.

[34] J. Zobel. Derivation of polymorphic types for Prolog programs. In J.-L. Lassez, editor, *Logic Programming: Proceedings of the fourth international conference*, pages 817–838, Australia, 1987.

A Appendix

This appendix contains those proofs for the lemmas and the theorem that do not immediately follow from the definitions and the lemmas appearing in the paper.

Let t be a term. The height $h(t)$ of t is defined as follows.

$$h(t) \stackrel{def}{=} \begin{cases} 0, & if\ t\ is\ a\ constant\ or\ a\ variable \\ 1 + max\{h(t_1), \cdots, h(t_n)\}, & \\ & otherwise, t = f(t_1, \cdots, t_n) \end{cases}$$

Proof of lemma 3.7 (p. 3): Equation 1 can be written as $\check{\gamma}(M) = \bigcup_{k \geq 0}\{\check{\gamma}_k(M)\}$ where

$$\check{\gamma}_k(M) = \begin{cases} \mathcal{TERMS} & if\ M = \check{\top} \\ \emptyset & if\ M = \check{\bot} \vee k = 0 \\ \left\{ f(t_1, \cdots, t_n) \,\middle|\, \begin{array}{c} f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow \\ X_1 : T_1, \cdots, X_n : T_n \\ \kappa = \{\beta_j/M_j | 1 \leq j \leq m\} \\ \forall 1 \leq i \leq n.(t_i \in \check{\gamma}_{k-1}(T_i\kappa)) \end{array} \right\} & \\ & if\ M = c(M_1, \cdots, M_m) \end{cases}$$

It suffices to prove that, for any $k \geq 0$,

$$M \check{\sqsubseteq} M' \rightarrow \check{\gamma}_k(M) \subseteq \check{\gamma}_k(M') \tag{15}$$

If either $M = \check{\bot}$ or $M' = \check{\top}$ then equation 15 holds for any $k \geq 0$. The proof for the case where $M \neq \check{\bot}$ and $M' \neq \check{\top}$ is done by induction on k.

Basis. $\check{\gamma}_0(M) = \check{\gamma}_0(M') = \emptyset$. So, equation 15 holds for $k = 0$.

Induction. If $M \check{\sqsubseteq} M'$ then, by definition 3.1, there are a type constructor c/m and mono-types $M_1, \cdots, M_m$, $M'_1, \cdots, M'_m$ such that $M = c(M_1, \cdots, M_m)$, $M' = c(M'_1, \cdots, M'_m)$ and $M_j \check{\sqsubseteq} M'_j$ for $1 \leq j \leq m$. Let $f(t_1, \cdots, t_n) \in \check{\gamma}_k(M)$ be an arbitrary term. It remains to prove that $f(t_1, \cdots, t_n) \in \check{\gamma}_k(M')$. Let

$$\begin{array}{r} f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow \\ X_1 : T_1, \cdots, X_n : T_n \end{array}$$

$t_i \in \check{\gamma}_{k-1}(T_i\{\beta_j/M_j \mid 1 \leq j \leq m\})$ since $f(t_1, \cdots, t_n) \in \check{\gamma}_k(M)$. Since $M_j \check{\sqsubseteq} M'_j$ for $1 \leq j \leq m$, $T_i\{\beta_j/M_j \mid 1 \leq j \leq m\} \check{\sqsubseteq} T_i\{\beta_j/M'_j \mid 1 \leq j \leq m\}$. Thus, $t_i \in \check{\gamma}_{k-1}(T_i\{\beta_j/M'_j \mid 1 \leq j \leq m\})$ according to the induction hypothesis. Therefore, $f(t_1, \cdots, t_n) \in \check{\gamma}_k(M')$. ∎

Proof of lemma 4.1 (p. 4): The proof is by induction on the height $h(t)$ of t.

Basis. $h(t) = 0$. t is either a constant or a variable. If t is a constant then $t\theta = t$. If t is a variable then $M = \check{\top}$ by equation 1. Therefore, $t \in \check{\gamma}(M) \rightarrow t\theta \in \check{\gamma}(M)$ when $h(t) = 0$.

Induction. $h(t) > 0$. There are a function symbol $f/n \in \Sigma$ and terms $t_1, \cdots, t_n$ such that $t = f(t_1, \cdots, t_n)$. Let

$$\begin{array}{r} f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow \\ X_1 : T_1, \cdots, X_n : T_n \end{array}$$

By equation 1, there are $M_1, \cdots, M_m$ such that $M = c(M_1, \cdots, M_m)$ and $t_i \in \check{\gamma}(T_i\{\beta_j/M_j | 1 \leq j \leq m\})$. $h(t_i) < h(t)$. $t_i\theta \in \check{\gamma}(T_i\{\beta_j/M_j \mid 1 \leq j \leq m\})$ by the induction hypothesis. Hence, $t\theta = f(t_1\theta, \cdots, t_n\theta) \in \check{\gamma}(M)$ by equation 1. Therefore, $t \in \check{\gamma}(M) \rightarrow t\theta \in \check{\gamma}(M)$ when $h(t) > 0$. ∎

Proof of lemma 4.2 (p. 4): The proof is by induction on the height $h(t)$ of t.

Basis. $h(t) = 0$. t must be a constant for otherwise $M = \check{\top}$ by equation 1.

Induction. $h(t) > 0$. There are a function symbol $f/n \in \Sigma$ and terms $t_1, \cdots, t_n$ such that $t = f(t_1, \cdots, t_n)$. Let

$$\begin{array}{r} f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow \\ X_1 : T_1, \cdots, X_n : T_n \end{array}$$

By equation 1, there are $M_1, \cdots, M_m$ such that $M = c(M_1, \cdots, M_m)$ and $t_i \in \check{\gamma}(T_i\{\beta_j/M_j \mid 1 \leq j \leq m\})$. $\check{\top}$ does not occur in $T_i\{\beta_j/M_j \mid 1 \leq j \leq m\}$ for $1 \leq i \leq n$ since $\check{\top}$ does not occur in M. $h(t_i) < h(t)$. By the induction hypothesis, t_i is a ground term for $1 \leq i \leq n$. Therefore, t is a ground term. ∎

Proof of lemma 4.9 (p. 6): The proof is done by induction on the height $h(t)$ of t.

Base. $h(t) = 0$ implies that t is either is variable or a constant. If t is a variable then $\mu(t) \check{\sqsubseteq} \nu(t)$ and, by equation 3, $\check{A}(t, \mu) \check{\sqsubseteq} \check{A}(t, \nu)$. If t is a constant then $\check{A}(t, \mu) = \check{A}(t, \nu)$. So, $\check{A}(t, \mu) \check{\sqsubseteq} \check{A}(t, \nu)$ for $h(t) = 0$.

Induction. $h(t) > 0$. Let $t = f(t_1, \cdots, t_n)$. By the induction hypothesis, $\forall 1 \leq i \leq n.\check{A}(t_i, \mu) \check{\sqsubseteq} \check{A}(t_i, \nu)$. Let

$$\begin{array}{r} f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow \\ X_1 : T_1, \cdots, X_n : T_n \end{array}$$

$$\kappa_i = lmts(\check{A}(t_i, \mu), T_i)$$

$$\kappa'_i = lmts(\check{A}(t_i, \nu), T_i)$$

If $\check{A}(t, \mu) = \check{\top}$ then $\exists 1 \leq i \leq n.\kappa_i = \mathit{fail}$ by equation 3. $\kappa'_i = \mathit{fail}$ by lemma 4.6.(b), $\check{A}(t, \nu) = \check{\top}$ by equation 3 and therefore $\check{A}(t, \mu) \check{\sqsubseteq} \check{A}(t, \nu)$ when $\check{A}(t, \mu) = \check{\top}$. If $\check{A}(t, \mu) \neq \check{\top}$ then $\forall 1 \leq i \leq n.\kappa_i \neq \mathit{fail}$ and $\check{A}(t, \mu) = c(\beta_1, \cdots, \beta_m)\kappa$ where

$$\kappa = \bigcup_{1 \leq j \leq m} \{\beta_j/(\check{\bigsqcup}_{\substack{1 \leq i \leq n \\ \beta_j \in dom(\kappa_i)}} \{\beta_j\kappa_i\})\}$$

If $\exists 1 \le i \le n.\kappa_i' = \mathit{fail}$ then $\check{A}(t,\nu) = \check{\top}$ and hence $\check{A}(t,\mu) \check{\sqsubseteq} \check{A}(t,\nu)$. Otherwise, $\check{A}(t,\nu) = c(\beta_1,\cdots,\beta_m)\kappa'$ where

$$\kappa' = \bigcup_{1\le j\le m} \{\beta_j/(\check{\bigsqcup}_{\substack{1\le i\le n\\ \beta_j\in dom(\kappa_i')}} \{\beta_j\kappa_i'\})\}$$

$\forall 1 \le j \le m.\beta_j\kappa \check{\sqsubseteq} \beta_j\kappa'$ since, according to lemma 4.6.(c), $\forall 1 \le i \le n.\forall\beta \in vars(T_i).\beta\kappa_i \check{\sqsubseteq} \beta\kappa_i'$. So, $\check{A}(t,\mu) \check{\sqsubseteq} \check{A}(t,\nu)$ when $\check{A}(t,\mu) \neq \check{\top}$. Therefore, $\check{A}(t,\mu) \check{\sqsubseteq} \check{A}(t,\nu)$ for $h(t) > 0$. ∎

Proof of lemma 4.11 (p. 6): Since both $t \in \check{\gamma}(M)$ and $\check{\alpha}(t) \check{\sqsubseteq} M$ hold when $M = \check{\top}$, we assume that $M \neq \check{\top}$ and accomplish the proof by induction on the height $h(t)$ of t.

Base. $h(t) = 0 \wedge M \neq \check{\top}$ implies that t is a constant. Let

$$t : c(\beta_1,\cdots,\beta_m) \leftarrow$$

Then

$$\begin{array}{rcll}\check{\alpha}(t) \check{\sqsubseteq} M & \leftrightarrow & c(\check{\bot},\cdots,\check{\bot}) \check{\sqsubseteq} M & \textit{by equation 2}\\ & \leftrightarrow & \exists M_1,\cdots,M_m.M = c(M_1,\cdots,M_m) & \\ & & & \textit{by definition 3.1}\\ & \leftrightarrow & t \in \check{\gamma}(M) & \textit{by equation 1}\end{array}$$

So, $t \in \check{\gamma}(M) \leftrightarrow \check{\alpha}(t) \check{\sqsubseteq} M$ when $h(t) = 0$.

Induction. $h(t) > 0$ implies that there are a function symbol $f/n \in \Sigma$ and terms $t_1,\cdots,t_n$ such that $t = f(t_1,\cdots,t_n)$. Let

$$\begin{array}{l}f(X_1,\cdots,X_n) : c(\beta_1,\cdots,\beta_m) \leftarrow\\ \qquad X_1 : T_1,\cdots,X_n : T_n\end{array}$$

$$\kappa_i = lmts(\check{\alpha}(t_i),T_i)$$

$$M = c(\beta_1,\cdots,\beta_m)\kappa''$$

We have that

$$\begin{array}{ll} t \in \check{\gamma}(M) & \\ \leftrightarrow \forall 1 \le i \le n.(t_i \in \check{\gamma}(T_i\kappa'')) & (\textit{by equation 1})\\ \leftrightarrow \forall 1 \le i \le n.\check{\alpha}(t_i) \check{\sqsubseteq} T_i\kappa'' & \\ & (\textit{by the induction hypothesis})\\ \leftrightarrow \forall 1 \le i \le n.(\kappa_i \neq \mathit{fail} \wedge \forall\beta \in vars(T_i).\beta\kappa_i \check{\sqsubseteq} \beta\kappa'') & \\ & (\textit{by lemma 4.6.(d)})\\ \leftrightarrow \forall 1 \le j \le m.(\check{\bigsqcup}_{\substack{1\le i\le n\\ \beta_j\in dom(\kappa_i)}} \{\beta_j\kappa_i\} \check{\sqsubseteq} \beta_j\kappa'') & \\ \leftrightarrow \check{\alpha}(t) \check{\sqsubseteq} M & (\textit{by equation 2})\end{array}$$

Therefore, $t \in \check{\gamma}(M) \leftrightarrow \check{\alpha}(t) \check{\sqsubseteq} M$ when $h(t) > 0$. ∎

Proof of lemma 4.14 (p. 6): If $M = \check{\top}$ then $\forall X \in vars(t).\nu(X) = \check{\top}$ by equation 4 and hence $\forall X \in vars(t).(\check{\alpha}(X\theta) \check{\sqsubseteq} \nu(X))$. The proof for the case $M \neq \check{\top}$ is done by induction on the height $h(t)$ of t.

Basis. $h(t) = 0$ implies either t is a variable or t is a constant. If t is a variable then $\nu(t) = M$ by equation 4 and hence $\check{\alpha}(t\theta) \check{\sqsubseteq} M \rightarrow \forall X \in vars(t).(\check{\alpha}(X\theta) \check{\sqsubseteq} \nu(X))$. If t is a constant then $vars(t) = \emptyset$ and therefore $\forall X \in vars(t).\check{\alpha}(X\theta) \check{\sqsubseteq} \nu(X)$. So,

$$\check{\alpha}(t\theta) \check{\sqsubseteq} M \rightarrow \forall X \in vars(t).\check{\alpha}(X\theta) \check{\sqsubseteq} \nu(X)$$

when $h(t) = 0$.

Induction. $h(t) > 0$ implies that there are a function symbol $f/n \in \Sigma$ and terms $t_1,\cdots,t_n$ such that $t = f(t_1,\cdots,t_n)$. Let

$$\begin{array}{l}f(X_1,\cdots,X_n) : c(\beta_1,\cdots,\beta_m) \leftarrow\\ \qquad X_1 : T_1,\cdots,X_n : T_n\end{array}$$

$$\kappa_i = lmts(\check{\alpha}(t_i\theta),T_i)$$

$$\kappa = \bigcup_{1\le j\le m} \{\beta_j/(\check{\bigsqcup}_{\substack{1\le i\le n\\ \beta_j\in dom(\kappa_i)}} \{\beta_j\kappa_i\})\}$$

Since $M \neq \check{\top}$, $\check{\alpha}(t\theta) = c(\beta_1,\cdots,\beta_m)\kappa$ and $\kappa_i \neq \mathit{fail}$ for $1 \le i \le n$ by equations 2 and 3. $M \neq \check{\top}$ and $\check{\alpha}(t\theta) = c(\beta_1,\cdots,\beta_m)\kappa \check{\sqsubseteq} M$ imply that there is a type substitution κ' such that $M = c(\beta_1,\cdots,\beta_m)\kappa'$ and $\forall 1 \le j \le m.\beta_j\kappa \check{\sqsubseteq} \beta_j\kappa'$. Let

$$\nu_i = gvmt(T_i\kappa',t_i)$$

We have that

$$\begin{array}{ll}\kappa_i = lmts(\check{\alpha}(t_i\theta),T_i) & \\ \rightarrow \check{\alpha}(t_i\theta) \check{\sqsubseteq} T_i\kappa_i & \textit{by lemma 4.6.(a)}\\ \rightarrow \check{\alpha}(t_i\theta) \check{\sqsubseteq} T_i\kappa & \textit{by definition of } \kappa\\ \rightarrow \check{\alpha}(t_i\theta) \check{\sqsubseteq} T_i\kappa' & \textit{by the transitivity of } \check{\sqsubseteq}\\ \rightarrow \forall X \in vars(t_i).(\check{\alpha}(X\theta) \check{\sqsubseteq} \nu_i(X)) & \\ & \textit{by the induction hypothesis}\\ \rightarrow \forall X \in vars(t).(\check{\alpha}(X\theta) \check{\sqsubseteq} \nu(X)) & \textit{by equation 4}\end{array}$$

So, $\check{\alpha}(t\theta) \check{\sqsubseteq} M \rightarrow \forall X \in vars(t).(\check{\alpha}(X\theta) \check{\sqsubseteq} \nu(X))$ when $h(t) > 0$, according to equation 2. ∎

Proof of lemma 4.16 (p. 6): A variable mono-typing whose domain is $\mathcal{V}$ is an abstract instantiation state on $\mathcal{V}$. Let

$$\mathcal{I}_1 = gvmt(\mathcal{I}(X),t)$$

and

$$\mathcal{I}_2 = \{Y/(\mathcal{I}(Y)\check{\sqcap}\mathcal{I}_1(Y)) \mid Y \in vars(t)\}))$$

Since $\sigma \in \gamma_i[\![\mathcal{V}]\!](\mathcal{I})$, $X\sigma \in \check{\gamma}(\mathcal{I}(X))$ and $Y\sigma \in \check{\gamma}(\mathcal{I}(Y))$ for $Y \in vars(t)$. By lemma 4.1, $t\sigma\eta = X\sigma\eta \in \check{\gamma}(\mathcal{I}(X))$. Therefore, $Y\sigma\eta \in \check{\gamma}(\mathcal{I}_1(Y))$ for $Y \in vars(t)$ by lemmas 4.11 and 4.14. By lemma 4.1, $Y\sigma\eta \in \check{\gamma}(\mathcal{I}(Y))$. Therefore, by lemma 4.14 and equation 5, $Y\sigma\eta \in \check{\gamma}(\mathcal{I}_2(Y)) = \check{\gamma}(\mathcal{I}_0(Y))$. By lemma 4.10.(a) and lemma 4.14, $X\sigma\eta = t\sigma\eta \in \check{\gamma}(\check{A}(t,\mathcal{I}_2))$. So, $X\sigma\eta \in \check{\gamma}(\mathcal{I}(X)\check{\sqcap}\check{A}(t,\mathcal{I}_2)) \subseteq \check{\gamma}(\mathcal{I}_0(X))$ by equation 5. ∎

Proof of theorem 5.2 (p. 8): We first give the following two lemmas that are used in the proof of the theorem.

Lemma A.1 *If*

$$soln(X = t, <\mathcal{I}_i,\mathcal{S}_i,\mathcal{A}_i>) = <\mathcal{I}_o,\mathcal{S}_o,\mathcal{A}_o>$$

$$\sigma \in \gamma[\![\mathcal{W}]\!](<\mathcal{I}_i,\mathcal{S}_i,\mathcal{A}_i>)$$

and

$$\eta = \sigma \circ mgu(\{X\sigma = t\sigma\}) \neq \mathit{fail}$$

then $\eta \in \gamma[\![\mathcal{W}]\!](<\mathcal{I}_o,\mathcal{S}_o,\mathcal{A}_o>)$.

Proof: $soln(X = t, <\mathcal{I}_i,\mathcal{S}_i,\mathcal{A}_i>) = <\mathcal{I}_o,\mathcal{S}_o,\mathcal{A}_o>$, by equation 11, if

$$unieq(X = t, <\mathcal{I}_i,\mathcal{S}_i,\mathcal{A}_i>) = <\mathcal{I}',\mathcal{S}',\mathcal{A}'>$$

$$compose(<\mathcal{I}_i,\mathcal{S}_i,\mathcal{A}_i>, <\mathcal{I}',\mathcal{S}',\mathcal{A}'>) = <\mathcal{I}'',\mathcal{S}'',\mathcal{A}''>$$

and

$$reduce(<\mathcal{I}'', \mathcal{S}'', \mathcal{A}''>) = <\mathcal{I}_o, \mathcal{S}_o, \mathcal{A}_o>$$

$\gamma[\![\mathcal{W}]\!](<\mathcal{I}'', \mathcal{S}'', \mathcal{A}''>) = \gamma[\![\mathcal{W}]\!](<\mathcal{I}_o, \mathcal{S}_o, \mathcal{A}_o>)$ by equations 14 and lemma 4.2. So, it is sufficient to prove that, for any $U, V \in \mathcal{W}$,

(a) if $U\eta \neq V\eta$ then $(U, V) \notin \mathcal{A}''$ and

(b) if $U \neq V \wedge vars(U\eta) \cap vars(U\eta) \neq \emptyset$ then $(U, V) \in \mathcal{S}''$ and if $\chi(U\eta) = 2$ then $(U, U) \in \mathcal{S}''$ and

(c) $U\eta \in \check{\gamma}(\mathcal{I}''(U))$.

(a) If $(U, V) \in \mathcal{A}''$ then either (a.1) $(U, V) \in \mathcal{A}_i$ or (a.2) $t \in \mathcal{W}$ and $\exists W, Z \in \{X, t\}.((U, W) \in \mathcal{A}_i \wedge (Z, V) \in \mathcal{A}_i)$. In case (a.1), $U\sigma = V\sigma$ and hence $U\eta = V\eta$. In case (a.2), $U\sigma = W\sigma$ and $V\sigma = Z\sigma$, so, $U\eta = V\eta$ because $\eta = \sigma \circ mgu(\{X\sigma = t\sigma\})$ and $W, Z \in \{X, t\}$.

(b) results from Lemma 2.2.(3)-(5) and Lemma 4.4.(2) in [6]. The part of our abstract unification function devoted to sharing analysis is essentially the same as that in [6].

(c) By lemmas 4.16 and 4.1 and equation 13, $U\eta \in \check{\gamma}(\mathcal{I}''(U))$ for any variable $U \in \{X\} \cup vars(t)$. ∎

Lemma A.2 *Let $\sigma^\flat \in Sub^\flat[\![\mathcal{W}]\!]$ and E be a set of equations in solved form such that $vars(E) \subseteq \mathcal{W}$. If $\sigma \in \gamma[\![\mathcal{W}]\!](\sigma^\flat)$ and $mgu(E\sigma) \neq fail$ then $\sigma \circ mgu(E\sigma) \in \gamma[\![\mathcal{W}]\!](solve(E, \sigma^\flat))$.*

Proof: The proof is done by induction on the size of E. If $E = \emptyset$ then $\sigma \circ mgu(E\sigma) = \sigma \circ \emptyset = \sigma \in \gamma[\![\mathcal{W}]\!](\sigma^\flat) = \gamma[\![\mathcal{W}]\!](solve(E, \sigma^\flat))$ by equation 8.

Suppose $E \neq \emptyset$. Let $E = \{X = t\} \cup E'$ and $\eta = mgu(X\sigma, t\sigma)$. Then $\sigma \circ \eta \in \gamma[\![\mathcal{W}]\!](soln(X = t, \sigma^\flat))$ by lemma A.1. Let $\zeta = mgu(E'(\sigma \circ \eta))$. By the induction hypothesis and equation 8,

$$\begin{aligned}\sigma \circ \eta \circ \zeta &\in \gamma[\![\mathcal{W}]\!](solve(E', soln(X = t, \sigma^\flat))) \\ &= \gamma[\![\mathcal{W}]\!](solve(E, \sigma^\flat))\end{aligned}$$

$\sigma \circ \eta \circ \zeta = \sigma \circ (\eta \circ \zeta) = \sigma \circ mgu(E\sigma)$. So, $\sigma \circ mgu(E\sigma) \in \gamma[\![\mathcal{W}]\!](solve(E, \sigma^\flat))$. ∎

We now prove the theorem. Let Ψ be also a renaming substitution such that $vars(A\theta\Psi) \cap vars(B\sigma) = \emptyset$. Let $\mathcal{W} = \mathcal{U}\Psi \cup \mathcal{V}$, $\eta = \sigma \cup \theta\Psi$ and $E_0 = eq(mgu(\{A\Psi = B\}))$. $\zeta \in \gamma[\![\mathcal{W}]\!](\zeta^\flat) \leftrightarrow \zeta\!\upharpoonright\!\mathcal{W} \in \gamma[\![\mathcal{W}]\!](\zeta^\flat)$ for any $\zeta \in Sub$ and $\zeta^\flat \in Sub^\flat[\![\mathcal{W}]\!]$. Therefore,

$$\begin{aligned}
& unify(A, \theta, B, \sigma) \in \gamma[\![\mathcal{V}]\!](unify^\flat(A, \theta^\flat, B, \sigma^\flat)) \\
\leftrightarrow\ & \sigma \circ mgu(\{(A\Psi)(\theta\Psi) = B\sigma\}) \\
& \in \gamma[\![\mathcal{V}]\!](unify^\flat(A, \theta^\flat, B, \sigma^\flat)) \\
\leftrightarrow\ & (\sigma \cup \theta\Psi) \circ mgu(\{(A\Psi)(\sigma \cup \theta\Psi) = B(\sigma \cup \theta\Psi)\}) \\
& \in \gamma[\![\mathcal{V}]\!](unify^\flat(A, \theta^\flat, B, \sigma^\flat)) \\
\leftrightarrow\ & \eta \circ mgu(E_0\eta) \in \gamma[\![\mathcal{V}]\!](unify^\flat(A, \theta^\flat, B, \sigma^\flat)) \qquad (16)
\end{aligned}$$

By equation 7, $\eta \in \gamma[\![\mathcal{W}]\!](\theta^\flat\Psi \uplus \sigma^\flat)$. By lemma A.2,

$$\eta \circ mgu(E_0\eta) \in \gamma[\![\mathcal{W}]\!](solve(E_0, \theta^\flat\Psi \uplus \sigma^\flat))$$

So, $unify(A, \theta, B, \sigma) \in \gamma[\![\mathcal{V}]\!](unify^\flat(A, \theta^\flat, B, \sigma^\flat))$ by equations 6, 10 and 16. ∎