Path Dependent Analysis of Logic Programs*

Lunjin Lu
Department of Computer Science and Engineering
Oakland University
Michigan 48309, USA
EMail: L2LU@oakland.edu
http://www.oakland.edu/~L2LU

ABSTRACT

This paper presents an abstract semantics that uses information about execution paths to improve precision of data flow analyses of logic programs. We illustrate the abstract semantics by abstracting execution paths using call strings of fixed length and the last transfer of control. Abstract domains that have been developed for logic program analyses can be used with the new abstract semantics without modification.

Keywords: Abstract interpretation, Context sensitive analysis, Call strings

1. INTRODUCTION

Abstract interpretation [8] is a program analysis methodology for statically deriving run-time properties of programs. The derived program properties are then used by other program manipulation tools such as compilers and partial evaluators. Program analyses are viewed as program executions over non-standard data domains. The idea is to define a collecting semantics for a program which associates with each program point the set of the states that are obtained whenever the execution reaches the point. Then an approximation of the collecting semantics is calculated by simulating over a non-standard data domain (called the abstract domain) the computation of the collecting semantics over the standard data domain (called the concrete domain).

There has been much research into abstract interpretation of logic programs [9]. A number of (generic) abstract semantics, often called frameworks, schemes [3, 4, 18, 34, 31], have been proposed for abstract interpretation of logic programs. These abstract semantics have been specialized for the detection of determinacy [10], data dependency analyses [11, 16], mode inference [11, 39], program transformation [35], type inference [17], termination proof [41], etc. Since analysis of logic programs is inherently interprocedural, it is natural to make use of information about the context of invocations to improve analysis. In logic program analysis, information about the context of a call has been exclusively captured by recording information about the state in which the call was made. No abstract semantics for logic programs make use of call strings as context information. This paper fills this gap by deriving an abstract semantics that is parameterized by both an abstraction of execution paths and an abstraction of data.

The remainder of this paper is organized as follows. Section 2 briefly recalls on terminology in logic programming, and introduces some notations used later in this paper. Section 3 reformulates SLD with the left-to-right computation rule in order to facilitate the derivation of a collecting semantics. In the sequel, We will omit reference to the left-to-right computation rule. Section 4 derives the collecting semantics from the operational semantics. Section 5 derives the abstract semantics from the collecting semantics, and gives the sufficient conditions for the abstract semantics to approximate safely the collecting semantics. In section 6, we show how the abstract semantics can be specialized by two examples. The first example uses call strings of length one as context information and the second example uses the last transfer of control as context information. Section 7 reviews related work and section 8 concludes. Only definite programs are considered in this paper. However, the abstract semantics can be readily generalized to analyze logic programs with negation and builtin predicates as in [3]. Proofs are included in an appendix.

2. PRELIMINARIES

The reader is assumed to be familiar with the terminology of logic programming [24] and that of abstract interpretation [8].

Let Σ be a set of *function symbols*, Π a set of *predicate symbols* and $\mathcal{VAR}$ a denumerable set of variables. Let $\mathcal{U} \subseteq \mathcal{VAR}$. $\mathsf{Term}(\Sigma, \mathcal{U})$ denotes the set of *terms* that can be constructed from Σ and $\mathcal{U}$. $\mathsf{Atom}(\Sigma, \Pi, \mathcal{U})$ is the set of *atoms* that is con-

PEPM '02, Jan. 14-15, 2002 Portland, OR, USA ©2002 ACM ISBN 1-58113-455-X/02/01...$5.00

structible from Π and $\mathsf{Term}(\Sigma, \mathcal{U})$. $\mathsf{Term} \stackrel{def}{=} \mathsf{Term}(\Sigma, \mathcal{VAR})$ and $\mathsf{Atom} \stackrel{def}{=} \mathsf{Atom}(\Pi, \Sigma, \mathcal{VAR})$. Let θ be a *substitution*. Then $dom(\theta)$ denotes the domain of θ. The identity substitution is denoted by ϵ.

An *equation* is a formula of the form $l = r$ where either $l, r \in \mathsf{Term}$ or $l, r \in \mathsf{Atom}$. The set of all equations is denoted as Eqn. For a set of equations $E \in \wp(\mathsf{Eqn})$, $mgu : \wp(\mathsf{Eqn}) \mapsto Sub \cup \{\mathsf{fail}\}$ returns either a most general unifier for E if E is unifiable or fail otherwise, where Sub is the set of idempotent substitutions. $mgu(\{l = r\})$ is sometimes written as $mgu(l, r)$. The function composition operator $\circ$ is defined as $f \circ g \stackrel{def}{=} \lambda x.f(g(x))$. Let $\theta \circ \mathsf{fail} \stackrel{def}{=} \mathsf{fail}$ and $\mathsf{fail} \circ \theta \stackrel{def}{=} \mathsf{fail}$ for any $\theta \in Sub \cup \{\mathsf{fail}\}$. We sometimes use Church's lambda notation for functions, so that a function f will be denoted by $\lambda x.f(x)$. Let $A, B \in \mathsf{Atom}$, and $\theta, \omega \in Sub$. Define

$$uf(A, \theta, B, \omega) \stackrel{def} {=} \begin{cases} \textit{let } \rho \textit{ be a renaming such that} \\ \quad vars(\rho(\theta(A))) \cap vars(\omega(B)) = \emptyset, \\ \textit{in} \\ mgu(\rho(\theta(A)), \omega(B)) \circ \omega \end{cases} \quad (1)$$

A *clause* C is a formula of the form $H \leftarrow A_1, A_2, \cdots, A_n$ where $H \in \mathsf{Atom}$ and $A_i \in \mathsf{Atom}$ for $1 \leq i \leq n$. H is called the *head* of the clause and $A_1, A_2, \cdots, A_n$ the *body* of the clause. We designate C with $n + 1$ different *program points* $p_1, p_2, \cdots p_{n+1}$ with point p_j immediately before A_j for $1 \leq j \leq n$ and point p_{n+1} immediately after A_n. $entry(C) \stackrel{def}{=} p_1$ is called the *entry point* of C and $exit(C) \stackrel{def}{=} p_{n+1}$ the *exit point* of C. A *goal* is a formula of the form $\leftarrow A_1, A_2, \cdots, A_n$ with $A_i \in \mathsf{Atom}$ for $1 \leq i \leq n$. A *program* is a set $\{C_\imath \mid \imath \in \Im_C\}$ of clauses where $\Im_C$ is a finite set of natural numbers. A *query* to a program is a goal that initiates the execution of that program. We designate a query with program points in the same way. There might be infinite number of possible queries that a program is intended to respond to. Let $\{G_k\Theta_{p_k} \mid k \in \Im_G\}$ be the set of all possible queries where $\Im_G$ is a finite set of natural numbers such that $\Im_G \cap \Im_C = \emptyset$, G_k for $k \in \Im_G$ is a goal, Θ_{p_k} is a set of substitutions and $p_k = entry(G_k)$. Each $G_k\theta_k$ with $\theta_k \in \Theta_{p_k}$ is a query. Let $\Im \stackrel{def}{=} \Im_C \cup \Im_G$. P_i refers to C_i for $i \in \Im_C$ and to G_i for $i \in \Im_G$. Let p be a program point. We write A_p to denote the atom to the right of p if p is not an exit point. If p is in a clause, we also write H_p to denote the head of the clause. $\mathcal{V}_p$ denotes the set of variables of interest at point p. $\mathcal{V}_p$ is usually the set of variables occurring in the clause in which p appears.

We denote by $\mathcal{N}$ the set of all program points designated with P_i for all $i \in \Im$. We use p^- to denote the program point to the left of p if p^- exists. Similarly, p^+ denotes the program point to the right of p if p^+ exists. We shall use $\mathcal{N}^0$ to denote the set of entry points of queries, $\mathcal{N}^1$ the set of entry points of clauses and $\mathcal{N}^2$ the set of all other program points.

Let $p, q \in \mathcal{N}$, and q be the most recent program point that SLD has reached. There are two possibilities that SLD will reach p next. If q is the exit point of a clause then SLD can reach p immediately only if that program clause has been used to resolve with A_{p^-}. If q is not the exit point of a clause and p is the entry point of another clause then SLD may reach p immediately by invoking that clause. Note that if q is the exit point of a query then SLD has succeeded and will not visit any more program points. We use a graph $< \mathcal{N}, \mathcal{E} >$, called program graph, to represent the relation among program points p and q that "SLD will possibly visit p immediately after it has visited q". Formally, $\mathcal{E}$ is defined as follows.

$$\begin{aligned} \mathcal{E} &\stackrel{def}{=} \bigcup_{1 \leq \jmath \leq 2} \mathcal{E}^\jmath \\ \mathcal{E}^1 &\stackrel{def}{=} \left\{ (entry(C), q) \;\middle|\; \begin{array}{l} C \in P \\ uf(A_q, \epsilon, H_{entry(C)}, \epsilon) \neq \mathsf{fail} \end{array} \right\} \\ \mathcal{E}^2 &\stackrel{def}{=} \{(p, exit(C)) \mid (entry(C), p^-) \in \mathcal{E}^1\} \end{aligned}$$

Edges in $\mathcal{E}^1$ correspond to procedure-entries, and edges in $\mathcal{E}^2$ to procedure-exits. We have $\mathcal{E}^1 \cap \mathcal{E}^2 = \emptyset$. Note that (p, q) is an edge from q to p.

EXAMPLE 2.1. *Consider the following logic program. The meaning of* $member(X, L)$ *is that* X *is a member of list* L*. The meaning of* $both(X, L, K)$ *is that* X *is a member of both list* L *and list* K*.*

$$\begin{aligned} C_1 &\equiv both(X, L, K) \leftarrow \textcircled{1} member(X, L), \textcircled{2} \\ &\qquad\qquad\qquad\qquad\quad member(X, K) \textcircled{3} \\ C_2 &\equiv member(X, [X|L]) \leftarrow \textcircled{4} \\ C_3 &\equiv member(X, [H|L]) \leftarrow \textcircled{5} member(X, L) \textcircled{6} \\ G_4 &\equiv \leftarrow \textcircled{7} both(X, L1, L2) \textcircled{8} \end{aligned}$$

Suppose that the set of queries is described by $\{G_4\Theta_7\}$ *with* Θ_7 *being the set of substitutions* θ *such that* $\theta(X)$ *is a variable, and both* $\theta(L)$ *and* $\theta(K)$ *are ground terms. Then,* $\Im_C = \{1, 2, 3\}$ *and* $\Im_G = \{4\}$*.* $\mathcal{N}$ *contains 8 program points.* $A_1 = member(X, L)$ *and* $A_2 = member(X, K)$*.* $\mathcal{V}_1 = \{X, L, K\}$ *and* $\mathcal{V}_7 = \{X, L1, L2\}$*.* $(5, 1)$ *is an edge in* $\mathcal{E}^1$ *and* $(2, 6)$ *is an edge in* $\mathcal{E}^2$*.* □

A path is either Λ denoting the empty path or a sequence of program points $p_n p_{n-1} \cdots p_1$ such that $(p_i, p_{i-1}) \in \mathcal{E}$ for $2 \leq i \leq n$ and $p_1 \in \mathcal{N}^0$. Note that p_1 is the starting point of the path. We use δ, χ and ϕ to denote paths and Δ to denote the set of all paths. Let $\Delta(p) \stackrel{def}{=} \{p\delta \mid p\delta \in \Delta\}$. Then $\Delta(p)$ is the set of paths leading to p (including p).

We define closed path segments as follows.

- δ is a closed path segment for A_p if $\delta = p^+\delta' p$ and δ' is a closed path segment for the body of a clause C and $(entry(C), p) \in \mathcal{E}$.
- δ is a closed path segment for the body $A_{p_1}, \cdots, A_{p_n}$ of a clause if $\delta = \delta_n \cdots \delta_1$ such that δ_i is a closed path segment for A_{p_i} for each $1 \leq i \leq n$. Note that $\delta = \Lambda$ when $n = 0$.

R(1)	*If*	$\begin{pmatrix} (p,q) \in \mathcal{E}^1 \\ \wedge \quad \theta = uf(A_q, \sigma, H_p, \epsilon) \neq \mathsf{fail} \end{pmatrix}$
	then	$(q\delta', \sigma)S \overset{P}{\rightsquigarrow} (pq\delta', \theta)(q\delta', \sigma)S$
R(2)	*If*	$\begin{pmatrix} (p,q) \in \mathcal{E}^2 \\ \wedge \quad \theta = uf(H_q, \sigma, A_{p^-}, \omega) \neq \mathsf{fail} \end{pmatrix}$
	then	$(q\delta' p^- \delta'', \sigma)(p^- \delta'', \omega)S \overset{P}{\rightsquigarrow} (pq\delta' p^- \delta'', \theta)S$

Figure 1: Transition rules

Let Δ_c be the set of the closed path segments for the atoms in the program and the bodies of the clauses in the program.

3. OPERATIONAL SEMANTICS

We use a variant of SLD resolution via the left to right computation rule (VSLD in abbreviation) as the operational semantics. VSLD differs from SLD in several ways. Firstly, a goal in VSLD is associated with its derivation path from the query. Secondly, when VSLD derives a new goal from the current goal and a clause, it renames the leftmost atom in the goal instead of the clause. This is to ensure that the domain of the substitution that will be applied to the body of the clause contains variables in the clause instead of their renamed counterparts. Thirdly, when a sub-refutation is finished, an extra renaming and an extra unification are needed for VSLD to calculate the substitution immediately after the sub-refutation whilst these extra operations are not needed in SLD.

A state in VSLD is a stack that is a sequence of stack items ending in a special symbol \$ denoting the empty stack. A stack item is of the form (δ, θ) where $\delta \in \Delta$ and $\theta \in Sub$. The set of all possible stacks is then

$$\mathcal{S} = (\Delta \times Sub)^* \times \{\$\}$$

where $\cdot^*$ is the Kleene closure operator.

VSLD is given by transition rules in Figure 1. Rule (1) performs a *procedure-entry* and rule(2) does a *procedure-exit*. The set $\mathcal{S}_0 \subseteq \mathcal{S}$ of initial states is determined by the set of queries to the program.

$$\mathcal{S}_0 \overset{def}{=} \{(p, \theta)\$ \mid p \in \mathcal{N}^0 \wedge \theta \in \Theta_p\}$$

The operational semantics of the program is defined as the set of descendant states of a set $\mathcal{S}_0$ of initial states below where $\overset{P}{\rightsquigarrow}_*$ is the reflexive and transitive closure of $\overset{P}{\rightsquigarrow}$.

$$[\![P]\!] \overset{def}{=} \{s \mid \exists s_0 \in \mathcal{S}_0.(s_0 \overset{P}{\rightsquigarrow}_* s)\}$$

LEMMA 3.1. *VSLD is equivalent to SLD in the sense that, given the same goal and the same program, VSLD reaches a program point iff SLD reaches the same program point, and the instantiation of the variables in the clause of the program point by VSLD is equivalent (modulo renaming) to that by SLD.* □

4. COLLECTING SEMANTICS

This section presents the collecting semantics. The collecting semantics first abstracts away the sequential relation between stack items of a stack and then classifies the stack items according to program points. It associates each point p with a mapping from a path δ ending at p to a set Θ of substitutions. Each point $p \in \mathcal{N}$ is thus associated with a member in $\Delta(p) \mapsto \wp(Sub)$. $< \wp(Sub), \subseteq, \emptyset, \cup >$ is a complete lattice. Therefore, the domain $\mathcal{D}^\sharp$ of the collecting semantics is the Cartesian product of the component domains $\Delta(p) \mapsto \wp(Sub)$ for all $p \in \mathcal{N}$. Let $X^\sharp \in \mathcal{D}^\sharp$. We use $X^\sharp(p)$ to denote the component in $X^\sharp$ that corresponds to point p. Let $X^\sharp, Y^\sharp \in \mathcal{D}^\sharp$ and define

$$X^\sharp \sqsubseteq^\sharp Y^\sharp \overset{def}{=} \forall p \in \mathcal{N}.\forall \delta \in \Delta(p).(X^\sharp(p)(\delta) \subseteq Y^\sharp(p)(\delta))$$

Then $< \mathcal{D}^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \sqcup^\sharp >$ is a complete lattice with $X^\sharp \sqcup^\sharp Y^\sharp = \lambda p \in \mathcal{N}.\lambda\delta \in \Delta(p).(X^\sharp(p)(\delta) \cup Y^\sharp(p)(\delta))$ and $\bot^\sharp = \lambda p \in \mathcal{N}.\lambda\delta \in \Delta(p).\emptyset$.

The approximation of a set of stacks by an element in $\mathcal{D}^\sharp$ is modeled by the following function $\gamma^\sharp \in \mathcal{D}^\sharp \mapsto \mathcal{D}$ where $suf(\delta)$ is the set of all suffixes of δ and $suf(\Phi) = \bigcup_{\phi \in \Phi} suf(\phi)$ for $\Phi \subseteq \Delta$.

$$\gamma^\sharp(X^\sharp) = \left\{ (p_n\delta_n, \theta_n) \cdots (p_1\delta_1, \theta_1)\$ \;\middle|\; \begin{array}{l} \forall 1 \leq i \leq n. \\ \quad (\theta_i \in X^\sharp(p_i)(p_i\delta_i)) \\ \wedge \\ \forall 1 \leq j < n. \\ \quad (p_j\delta_j \in suf(p_{j+1}\delta_{j+1})) \end{array} \right\} \tag{2}$$

$\gamma^\sharp$ is monotonic and $\gamma^\sharp(\mathcal{D}^\sharp)$ is a Moore family.

The collecting semantics is

$$[\![P]\!]^\sharp = lfp F_P^\sharp$$

where $F_P^\sharp : \mathcal{D}^\sharp \mapsto \mathcal{D}^\sharp$ is

$$F_P^\sharp(X^\sharp)(p)(\delta) \overset{def}{=}$$

$$\Theta_p \qquad \text{if } (p \in \mathcal{N}^0) \wedge (\delta = p) \tag{3}$$

$$uf^\sharp(A_q, X^\sharp(q)(q\delta'), H_p, \{\epsilon\}) \qquad \text{if } (p \in \mathcal{N}^1) \wedge (\delta = pq\delta') \tag{4}$$

$$uf^\sharp(H_q, X^\sharp(q)(q\delta' p^- \delta''), A_{p^-}, X^\sharp(p^-)(p^- \delta'')) \qquad \text{if } p \in \mathcal{N}^2 \wedge \delta = pq\delta' p^- \delta'' \tag{5}$$

and

$$uf^\sharp(A, \Theta, B, \Omega) \overset{def}{=} \{uf(A, \theta, B, \omega) \neq \mathsf{fail} \mid \theta \in \Theta \wedge \omega \in \Omega\} \tag{6}$$

for $A, B \in \mathsf{Atom}$ and $\Theta, \Omega \in \wp(Sub)$. $F_P^\sharp$ is a monotonic function on $< \mathcal{D}^\sharp, \sqsubseteq^\sharp >$. The correctness of $F_P^\sharp$ is given by the following lemma.

LEMMA 4.1. $[\![P]\!] \subseteq \gamma^\sharp([\![P]\!]^\sharp)$. □

5. ABSTRACT SEMANTICS

The collecting semantics $[\![P]\!]^\sharp$ is a safe approximation of the operational semantics and can be used as a basis for program analysis. $[\![P]\!]^\sharp(p)$ is a mapping from a path ending at p

to a set of substitutions. In order to obtain information effectively, further approximations are needed.

5.1 Abstracting paths

Paths of arbitrary length need be described by elements from a finite set $\Delta^\flat$. Elements in $\Delta^\flat$ are path descriptions. Let $\beta : \Delta \mapsto \Delta^\flat$ maps a path into its description. We require that

C0: $\beta : \Delta \mapsto \Delta^\flat$ is surjective, and if $\beta(q\delta) = \beta(q\delta')$ and $(p, q) \in \mathcal{E}$ then $\beta(pq\delta) = \beta(pq\delta')$

C0 ensures that each path description in $\Delta^\flat$ describes a non-empty set of paths and each execution step preserves the path equivalence induced by β. Define $\beta^{-1} : \Delta^\flat \mapsto \wp(\Delta)$ as $\beta^{-1}(\bar\delta) \stackrel{def}{=} \{\delta \mid \beta(\delta) = \bar\delta\}$. For any $\bar\delta \in \Delta^\flat$, $\{\beta(p\delta) \mid \delta \in \beta^{-1}(\bar\delta) \wedge p\delta \in \Delta\}$ is either $\{\bar\chi\}$ for some $\bar\chi \in \Delta^\flat$ or $\emptyset$. We use $p \bullet \bar\delta$ to denote $\bar\chi$ in the former case. We shall use $\bar\phi \ll \bar\chi$ to denote the condition that at least one path described by $\bar\phi$ is an extension of a path described by $\bar\chi$ with a closed path segment, i.e., $\bar\phi \ll \bar\chi \stackrel{def}{=} \exists\phi \in \beta^{-1}(\bar\phi).\exists\chi \in \beta^{-1}(\bar\chi).\exists\delta \in \Delta_c.(\chi = \delta\phi)$.

5.2 Abstracting data

When program is analyzed, the set of substitutions associated with a path ending at p is approximated by an abstract substitution associated with p. We follow [7] to parameterize abstract domains with finite sets of variables instead of having a single abstract domain for abstract substitutions associated with different program points or constructing abstract domains for different program points in different ways. Let $ASub_\mathcal{V}$ be the domain of abstract substitutions for $\mathcal{V}$ and $\gamma_\mathcal{V} \in ASub_\mathcal{V} \mapsto \wp(Sub)$ the function that gives meaning to an abstract substitution. Then $[\![P]\!]^\sharp(p)$ is described by a function from $\Delta^\flat(p)$ to $ASub_{\mathcal{V}_p}$ where $\Delta^\flat(p) = \{\beta(p\delta) \mid p\delta \in \Delta\}$ is the set of the descriptions of the paths ending at p. We require that, for any finite $\mathcal{V} \subseteq \mathcal{VAR}$,

C1: $< ASub_\mathcal{V}, \sqsubseteq_\mathcal{V}, \bot_\mathcal{V}, \sqcup_\mathcal{V} >$ is a complete lattice where $\sqsubseteq_\mathcal{V}$ is a partial order on $ASub_\mathcal{V}$, $\bot_\mathcal{V}$ the infimum and $\sqcup_\mathcal{V}$ the least upper bound operator; and

C2: $\gamma_\mathcal{V} \in ASub_\mathcal{V} \mapsto \wp(Sub)$ is monotonic and $\gamma_\mathcal{V}(ASub_\mathcal{V})$ is a Moore family.

The domain $\mathcal{D}^\flat$ of $F_P^\flat$ is constructed in the same way as the domain $\mathcal{D}^\sharp$ of $F_P^\sharp$. Each member $X^\flat$ in $\mathcal{D}^\flat$ is a vector that is indexed by program points. $X^\flat(p)$ is an element in $\Delta^\flat(p) \mapsto ASub_{\mathcal{V}_p}$. Let $X^\flat, Y^\flat \in \mathcal{D}^\flat$. Define

$$X^\flat \sqsubseteq^\flat Y^\flat \stackrel{def}{=} \forall p \in \mathcal{N}.\forall\bar\delta \in \Delta^\flat(p).(X^\flat(p)(\bar\delta) \ \sqsubseteq_{\mathcal{V}_p} \ Y^\flat(p)(\bar\delta))$$

$< \mathcal{D}^\flat, \sqsubseteq^\flat, \bot^\flat, \sqcup^\flat >$ is a complete lattice with

$$\begin{aligned} \bot^\flat &= \lambda p \in \mathcal{N}.\lambda\bar\delta \in \Delta^\flat(p).\bot_{\mathcal{V}_p} \\ X^\flat \sqcup^\flat Y^\flat &= \lambda p \in \mathcal{N}.\lambda\bar\delta \in \Delta^\flat(p).(X^\flat(p)(\bar\delta) \sqcup_{\mathcal{V}_p} Y^\flat(p)(\bar\delta)) \end{aligned}$$

The concretization function $\gamma^\flat : \mathcal{D}^\flat \mapsto \mathcal{D}^\sharp$ is defined in terms of $\gamma_{\mathcal{V}_p}$ and β. For every $X^\flat \in \mathcal{D}^\flat$,

$$\gamma^\flat(X^\flat) \stackrel{def}{=} \lambda p \in \mathcal{N}.\lambda\delta \in \Delta(p).\gamma_{\mathcal{V}_p}(X^\flat(p)(\beta(\delta))) \quad (7)$$

It follows from C2 that $\gamma^\flat$ is monotonic and $\gamma^\flat(\mathcal{D}^\flat)$ is a Moore family.

5.3 Abstract Semantics

The abstract semantics is obtained as follows. A set $\Theta \in \wp(Sub)$ of substitutions is replaced by an abstract substitution in $ASub_\mathcal{V}$ where $\mathcal{V}$ is the set of variables of interest. $uf^\sharp$ that is applied to two sets of substitutions described by $\theta^\flat \in ASub_\mathcal{U}$ and $\sigma^\flat \in ASub_\mathcal{V}$ respectively is replaced by $uf^\flat_{\mathcal{U},\mathcal{V}}$ that is applied to $\theta^\flat$ and $\sigma^\flat$. $\cup$ in the definition of $F_P^\sharp(X^\sharp)(p)$ is replaced by $\sqcup_{\mathcal{V}_p}$. Let $\theta^\flat_{p_k} \in ASub_{\mathcal{V}_{p_k}}$ be the least abstract substitution such that $\Theta_{p_k} \subseteq \gamma_{\mathcal{V}_{p_k}}(\theta^\flat_{p_k})$ for each $k \in \Im_G$. Note that $\theta^\flat_{p_k}$ instead of Θ_{p_k} is given before the program is analyzed. Let $\epsilon_{\mathcal{V}_p} \in ASub_{\mathcal{V}_p}$, called an abstract identity substitution in [3], be the least abstract substitution such that $\epsilon \in \gamma_{\mathcal{V}_p}(\epsilon_{\mathcal{V}_p})$ for each $p \in \mathcal{N}^1$. The abstract semantics is

$$[\![P]\!]^\flat = lfp F_P^\flat$$

where $F_P^\flat : \mathcal{D}^\flat \mapsto \mathcal{D}^\flat$ is

$$F_P^\flat(X^\flat)(p)(\bar\delta) \stackrel{def}{=}$$

$$\theta^\flat_p \qquad \text{if } (p \in \mathcal{N}^0) \wedge (\bar\delta = \beta(p)) \quad (8)$$

$$\sqcup_{\mathcal{V}_p} \left\{ uf^\flat_{\mathcal{V}_q,\mathcal{V}_p}(A_q, X^\flat(q)(\bar\chi), H_p, \epsilon_{\mathcal{V}_p}) \,\middle|\, \begin{array}{l}(p,q) \in \mathcal{E}^1 \\ \bar\delta = p \bullet \bar\chi\end{array} \right\} \quad \text{if } p \in \mathcal{N}^1 \quad (9)$$

$$\sqcup_{\mathcal{V}_p} \left\{ uf^\flat_{\mathcal{V}_q,\mathcal{V}_p}(H_q, X^\flat(q)(\bar\chi), A_{p^-}, X^\flat(p^-)(\bar\phi)) \,\middle|\, \begin{array}{l}(p,q) \in \mathcal{E}^2 \\ \bar\delta = p \bullet \bar\chi \\ \bar\phi \ll \bar\chi\end{array} \right\} \quad \text{if } p \in \mathcal{N}^2 \quad (10)$$

$F_P^\flat$ is a monotonic function on $< \mathcal{D}^\flat, \sqsubseteq^\flat >$. The following theorem establishes sufficient conditions for $lfp F_P^\flat$ to approximate correctly $lfp F_P^\sharp$.

THEOREM 5.1. $lfp F_P^\sharp \sqsubseteq^\sharp \gamma^\flat(lfp F_P^\flat)$ if C0-C4 hold where

C3: $\epsilon \in \gamma_\mathcal{V}(\epsilon_\mathcal{V})$.

C4: $uf^\sharp(A, \gamma_\mathcal{U}(\theta^\flat), B, \gamma_\mathcal{V}(\sigma^\flat)) \subseteq \gamma_\mathcal{V} \circ uf^\flat_{\mathcal{U},\mathcal{V}}(A, \theta^\flat, B, \sigma^\flat)$ for any finite $\mathcal{U}, \mathcal{V} \subseteq \mathcal{VAR}$, any $\theta^\flat \in ASub_\mathcal{U}$, any $\sigma^\flat \in ASub_\mathcal{V}$, and any atoms A and B such that $vars(A) \subseteq \mathcal{U}$ and $vars(B) \subseteq \mathcal{V}$.

□

We note that the conditions C1-C4 are exactly those required by the abstract semantics in [34, 25]. C1-C4 are conditions on the abstraction of data while C0 is a condition on the abstraction of paths. Once the abstraction β of paths is given, the abstract semantics is instantiated into a special form which can be used with an abstract domain satisfying C1-C4. Since these two abstractions are independent of each

other, abstract domains that have been designed for logic program analyses can be used with the abstract semantics without modification.

6. EXAMPLES

We now show the abstract semantics can be instantiated for different abstractions of paths. The simpliest abstraction of paths is to simply ignore them. This can be achieved by defining $\Delta^\flat \stackrel{def}{=} \mathcal{N}$ and $\beta(p\delta) \stackrel{def}{=} p$. Then $\Delta^\flat(p) = \{p\}$ and $p \bullet q = p$ if $(p,q) \in \mathcal{E}$ and $p^- \ll q$ if $(p,q) \in \mathcal{E}^2$. In this case, the abstract semantics degenerates to that in [34].

6.1 Call strings

Call strings have been used to enhance analysis of programs of other programming paradigms [36]. The idea is to keep track of calls on the execution stack - calls that are currently being executed. This amounts to ignore all segments of the execution path that correspond to those calls that has been fully executed. Let $call(p\delta)$ be the result of removing all closed path segment from δ. Since there might be stacks of infinite size due to recursion, it is usual to keep track of top k calls. This can be achieved by defining $\beta(\delta) \stackrel{def}{=} [call(\delta)]_k$ where $[\phi]_k$ be the path resulting from truncating ϕ at position $k+1$. Then $\Delta^\flat = \{\beta(\delta) \mid \delta \in \Delta\}$. If $p \in \mathcal{N}^1$ and $(p,q) \in \mathcal{E}^1$ and $\bar{\chi} \in \Delta^\flat(q)$ then $p \bullet \bar{\chi} = [p\bar{\chi}]_k$ since $(p,q) \in \mathcal{E}^1$ and $\bar{\chi} \in \Delta^\flat(q)$. Let $(p,q) \in \mathcal{E}^2$ and $\bar{\chi} \in \Delta^\flat(q)$. We have that $\bar{\delta} = p \bullet \bar{\chi}$ iff $\bar{\chi} = [p^-\bar{\delta}]_k$ and that $\bar{\phi} \ll \bar{\chi}$ implies $\bar{\phi} = \bar{\delta}$. Thus, $F^\flat_P$ is specialized into the following.

$$
\begin{array}{l}
F^\flat_P(X^\flat)(p)(\bar{\delta}) \stackrel{def}{=} \\
\quad \theta^\flat_p \qquad\qquad\qquad\qquad\qquad\qquad \text{if } (p \in \mathcal{N}^0) \wedge (\bar{\delta} = \Lambda) \\
\quad \sqcup_{\mathcal{V}_p} \left\{ uf^\flat_{\mathcal{V}_q,\mathcal{V}_p}(A_q, X^\flat(q)(\bar{\chi}), H_p, \epsilon_{\mathcal{V}_p}) \;\middle|\; \begin{array}{l}(p,q) \in \mathcal{E}^1 \\ \bar{\delta} = [p\bar{\chi}]_k\end{array} \right\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{if } p \in \mathcal{N}^1 \\
\quad \sqcup_{\mathcal{V}_p} \left\{ uf^\flat_{\mathcal{V}_q,\mathcal{V}_p}(H_q, X^\flat(q)(\bar{\chi}), A_{p^-}, X^\flat(p^-)(\bar{\delta})) \;\middle|\; \begin{array}{l}(p,q) \in \mathcal{E}^2 \\ \bar{\chi} = [p^-\bar{\delta}]_k\end{array} \right\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{if } p \in \mathcal{N}^2
\end{array}
$$

EXAMPLE 6.1. *Consider the program in 2.1 and call strings of length 1. Below is the result of mode analysis [3, 12] using above abstract semantics. The instantiation modes used are "free", "ground" and "top". A variable X is "free" in a substitution θ if $\theta(X)$ is a variable. X is "ground" in θ if $\theta(X)$ contains no variable. If the mode of X in θ is "top" then $\theta(X)$ can be any term. The analysis also keeps track of sharing [37] between variables to ensure correctness of analysis although no two variables in the same clause share in this example.*

```
$Goal :-
    % toplevel-[X/free,L1/ground,L2/ground],[]
  both(X,L1,L2)
    % toplevel-[X/ground,L1/ground,L2/ground],[]

member(X,[X|L]).
    % (both/3,1),1-[L/ground,X/ground],[]
    % (both/3,1),2-[L/ground,X/ground],[]
    % (member/2,2),1-[L/ground,X/ground],[]

member(X,[Y|L]) :-
    % (both/3,1),1-[L/ground,X/free,Y/ground],[]
    % (both/3,1),2-[L/ground,X/ground,Y/ground],[]
    % (member/2,2),1-[L/ground,X/top,Y/ground],[]
  member(X,L).
    % (both/3,1),1-[L/ground,X/free,Y/ground],[]
    % (both/3,1),2-[L/ground,X/ground,Y/ground],[]
    % (member/2,2),1-[L/ground,X/top,Y/ground],[]

both(X,L,K) :-
    % ($Goal/0,1),1-[X/free,L/ground,K/ground],[]
  member(X,L),
    % ($Goal/0,1),1-[X/ground,L/ground,K/ground],[]
  member(X,K).
    % ($Goal/0,1),1-[X/ground,L/ground,K/ground],[]
```

Each program point is annotated with a few comments. Each comment consists of a program point (which is the call string of length 1) and an abstract substitution. An abstract substitution has two parts. The first part represents mode information by assigning an instantiation mode to each variable of interest. The second part represents sharing information. A program point is represented by identifying the clause in which it appears and its position in the clause. A clause is identified by the name and arity of the predicate it defines and its textual position in the sequences of clauses for the predicate. For instance, ((member/2,2),1) stands for the entry point of the second clause defining the predicate member/2. A query is treated as a clause defining the predicate $Goal/0. A dummy call string toplevel indicates that the entry point of a query is reached by the language system.

The analysis result indicates that at the entry point of the second clause for member/2, X is a free variable if the clause is invoked at the point ((both/3,1),1) while X is a ground term if the clause is invoked at the point ((both/3,1),2). This information can be used to specialize member/2 into two different versions. Without keep tracking of path information, the two modes of X from these two different invocations must be merged resulting in the mode "top" which says nothing about the instantiation mode of X. □

6.2 Edges

Another useful abstraction of paths is to retain information about which clause is used to satisfy a given atom and which atom invokes a given clause. This corresponds to describing a path by its second element. Thus, $\beta(pq\delta) = q$ and $\beta(p) = \Lambda$. Note that $p \in \Delta$ implies $p \in \mathcal{N}^0$ and $q \in \Delta^\flat(p)$ implies $(p,q) \in \mathcal{E}$. Thus, $\Delta^\flat = \mathcal{N} \cup \{\Lambda\}$. It can be easily verified that C0 holds. We have $p \bullet \bar{\chi} = q$ for any $\bar{\chi} \in \Delta^\flat(q)$. We also have $\bar{\phi} \ll \bar{\chi}$ if $\phi \in \Delta^\flat(p^-)$, $\bar{\chi} \in \Delta^\flat(q)$ and $(p,q) \in \mathcal{E}^2$. Therefore, $F^\flat_P$ is specialized into the following.

$$F^\flat_P(X^\flat)(p)(q) \stackrel{def}{=}$$

$$\theta^\flat_p \qquad \text{if } (p \in \mathcal{N}^0) \wedge (q = \Lambda)$$

$$\sqcup_{\mathcal{V}_p}\{uf^\flat_{\mathcal{V}_q,\mathcal{V}_p}(A_q, X^\flat(q)(u), H_p, \epsilon_{\mathcal{V}_p}) \mid (q,u) \in \mathcal{E}\} \qquad \text{if } p \in \mathcal{N}^1$$

$$\sqcup_{\mathcal{V}_p}\left\{uf^\flat_{\mathcal{V}_q,\mathcal{V}_p}(H_q, X^\flat(q)(u), A_{p^-}, X^\flat(p^-)(v)) \middle| \begin{array}{l}(q,u) \in \mathcal{E} \\ (p^-,v) \in \mathcal{E}\end{array}\right\} \qquad \text{if } p \in \mathcal{N}^2$$

If we add Λ to $\mathcal{N}$ and make an edge from Λ to each point in $\mathcal{N}^0$, then the above abstract semantics associates an abstract substitution with each edge in the program graph.

EXAMPLE 6.2. *This example applies the above abstract semantics to perform prescriptive type analysis [19, 1, 20, 6, 26, 22]. In a prescriptive type analysis, type definitions are given as an analysis input. The following type definitions are used.*

$$\begin{array}{lcl} nat & ::= & 0 \mid s(nat) \\ list(\beta) & ::= & [\,] \mid [\beta|list(\beta)] \end{array}$$

Below is a buggy naive reverse program and the result of the prescriptive type analysis of the program using the abstract domain in [19]. The program is annotated as in the previous example. An abstract substitution is either vtbot *or a variable typing which is a mapping from a variable to a type.* vtbot *denotes the empty set of substitutions. A variable typing $\theta^\flat$ denotes the set of those substitutions that instantiate each variable X in the domain of $\theta^\flat$ into a term of the type $\theta^\flat(X)$.* bot *is the type denoting the empty set of terms and* top *is the type denoting the set of all terms.*

```
$Goal :-
    % toplevel - [X/list(nat)]
  nrev(X,Y).
    % (nrev/2,1),1 - [X/list(bot),Y/list(bot)]
    % (nrev/2,2),3 - [X/list(nat),Y/nat]

append([],L,L).
    % (append/3,2),1 - vtbot
    % (nrev/2,2),2 - [L/nat]
append([H|T],L,[H|TL]) :-
    % (append/3,2),1 - vtbot
    % (nrev/2,2),2 - vtbot
  append(T,L,TL).
    % (append/3,1),1 - vtbot
    % (append/3,2),2 - vtbot

nrev([],[]).
    % ($Goal/0,1),1 - []
    % (nrev/2,2),1 - []
nrev([H|T],L) :-
    % ($Goal/0,1),1 - [H/nat,T/list(nat)]
    % (nrev/2,2),1 - [H/nat,T/list(nat)]
  nrev(T,T1),
    % (nrev/2,1),1 -
    %          [H/nat,T/list(bot),T1/list(bot)]
    % (nrev/2,2),3 - [H/nat,T/list(nat),T1/nat]
  append(T1,H,L). % SHOULD BE append(T1,[H],L).
    % (append/3,1),1 -
    %    [H/nat,T/list(bot),L/nat,T1/list(bot)]
    % (append/3,2),2 - vtbot
```

The first comment for the exit point of the query tells that if the query is executed successfully with the first clause of the nrev/2 then both X and Y are instantiated into empty lists (of type list(bot)). This is expected. The second comment says that if the query is executed successfully with the second clause of the nrev/2 then X is instantiated into a list of natural numbers and Y into a natural number. This indicates that something is wrong with the second clause for nrev/2. The second comment for the exit point of second clause for nrev/2 says that the second clause for append/3 will fail when invoked by append(T1,H,L)*. The second comment for the entry point of the second clause for* append/3 *says that the unification will fail when the clause is invoked by* append(T1,H,L)*, indicating an error. Another indication of error is the second comment for the entry point of the first clause for append/3. It says that L will be a natural number instead of a list of natural numbers when the clause is invoked by* append(T1,H,L)*. Using the information, the bug can be easily located.*

The following is the result of the prescriptive type analysis by plugging the same abstract domain into the abstract semantics in [34] which ignores path information. The result is less precise than the above result. For instance, no type information is given for Y at the exit point of the query.

```
$Goal :-
        %[X/list(nat)],
    nrev(X,Y),
        %[X/list(nat)].

append([],L,L).
        %[L/nat].
append([H|T],L,[H|TL]) :-
        %[L/nat],
    append(T,L,TL).
        %[T/list(top),L/nat,TL/top].

nrev([],[])
        %[].
nrev([H|T],L) :-
        %[H/nat,T/list(nat)],
    nrev(T,T1),
        %[H/nat,T/list(nat),T1/top],
    append(T1,H,L).
        %[H/nat,T/list(nat),L/top,T1/list(top)].
```

Among other prescriptive type analyses of logic programs [1, 20, 6, 26, 22, 27], [27] is the most precise one. Using a disjunction of variable typings as an abstract substitution, [27] together with the abstract semantics in [34] infers that at the exit point of the query, either both X and Y are empty lists or X is of type list(nat) and Y of type nat. This information is precise so long as variables in the query are concerned. However, it does not tell which variable typing comes from which clause of nrev/2. □

7. RELATED WORK

Context information has been widely used in data flow analysis. For programs with high order constructs such as functional programs, information about contexts in which a procedure/function is applied may be obtained via a control flow analysis [33, 23]. Since only Horn clause logic programs are considered in our work, there is no need for a control flow analysis.

Context information has also been used in data flow analysis of logic programs. We now compare the abstract semantics proposed in this paper with other abstract semantics for logic programs. There are three approaches to abstract interpretation of logic programs. A top-down abstract semantics mimics a top-down evaluation strategy. A bottom-up abstract semantics approximates a bottom-up evaluation strategy. A fixed-point abstract semantics computes the least fixed-point of a system of simultaneous recurrence equations generated from the program.

7.1 Fixed-point abstract semantics

The abstract semantics in [31, 34] do not keep track of any context information at all. As shown in section 6, [34] is a special form of our abstract semantics. The abstract semantics in [30] records context information at the entry point of a program clause. Its abstract operators distinguish between different call instances. Since context information is not recorded at other program points, abstract substitutions originating from different clauses are merged together using the least upper bound operator. Our abstract semantics keeps track of more path information than [30] and therefore can infer more precise results. It also separates the abstraction of paths from that of data.

The abstract semantics in [43] approximates a minimum function graph semantics. A clause has as its denotation a partial function mapping an abstract substitution to another. Reachable versions of the predicates in the program are then computed from the abstract semantics where each reachable version of a predicate is a tuple of abstract substitutions one for each clause for the predicate. A compiler based on [43] may generate an implementation for each reachable version of the predicate. The correct version of a predicate is selected for a call in a version of a clause via an automaton whose states are reachable versions and whose inputs are call edges in the program graph. Context information is captured by reachable versions and the automaton. A set of paths is approximated by a regular set of call strings. Information about closed path segments is ignored that is useful as shown in example 6.2.

7.2 Bottom-up abstract interpretation

A bottom-up abstract semantics [2, 18, 5, 28, 29] approximates the success set of the program [40] using a bottom-up evaluation strategy. In order to infer call patterns, they first transform the program and then approximate the success set of the transformed program. Since there is no existing program transformation that encodes the execution path of the program, a bottom-up abstract semantics cannot make use of path information.

7.3 Top-down abstract interpretation

The abstract semantics in [3, 32, 42, 15] mimic SLD resolution. [32, 42, 15] differ from [3] only in their dealing with recursive calls. The abstract semantics in [3] constructs an abstract AND-OR graph that describes all the intermediate proof trees for the queries satisfying a query description. An AND-node is (labeled with) a clause head and its child OR-nodes are (labeled with) the atoms in the body of the clause. Every OR-node is adorned with its abstract call substitution and its abstract success substitution.

Consider an OR-node A with abstract call substitution $\theta^\flat$ in a partially constructed abstract AND-OR graph. The abstract semantics computes the abstract success substitution of A as follows. For each clause $H \leftarrow B_1, \cdots, B_m$ such that H may match with $\theta(A)$ for some θ satisfying $\theta^\flat$, it adds to A a child AND-node H that has m child OR-nodes $B_1, \cdots, B_m$ and performs an abstract procedure-entry to obtain the abstract call substitution $\theta^\flat_{in}$ of B_1. The abstract semantics extends B_1 recursively and extends B_{j+1} using the abstract success substitution of B_j as its abstract call substitution. After the abstract success substitution $\theta^\flat_{out}$ of the last OR-node B_m has been computed for each matching clause, the abstract success substitution $\eta^\flat$ of A is obtained by performing an abstract procedure-exit for each of these clauses and computing an upper bound of the results.

Suppose that an OR-node A with abstract call substitution $\theta^\flat$ were to be extended. If A has an ancestor OR-node A' with abstract call substitution $\eta^\flat$ such that A is a variant of A and $\theta^\flat$ is a variant of $\eta^\flat$, the abstract semantics initializes the abstract success substitution of A to the infimum abstract substitution and proceeds until the abstract success substitution of A' is computed. It then recomputes the part of the graph starting from the abstract success substitution of A to that of A' by using the abstract success substitution of A' as that of A. This is repeated until the abstract success substitution of A' stablizes. The same mechanism is also used to limit the size of the graph.

The context information captured in the abstract AND-OR graph is different from that in our abstract semantics. For a given program and an abstraction β of paths, our abstract semantics is instantiated into a fixed system of simultaneous recurrence equations. This is independent of the abstract domain and the abstract call substitution for the query. The shape of the abstract AND-OR graph depends on the abstract call substitution for the query. It decides how much context information is retained. Two variant atoms with variant abstract call substitutions or two atoms with the same predicate name and arity (when the depth of the abstract AND-OR graph exceeds some limit) are identified. This in a sense merges paths leading to different program points since these two atoms may appear in different places in the program. On the other hand, two paths leading to the same program point that have the same abstraction may be left un-merged. When the abstract success substitution of an OR-node A is computed, results of the procedure-exit operations are merged using an upper bound operator. This loses information about the closed execution paths for A.

The abstract semantics in [20, 4] mimic the OLDT resolution [38]. The comparison between our abstract semantics and an OLDT based abstract semantics is similar to that

between our abstract semantics and an abstract AND-OR graph based abstract semantics.

8. SUMMARY

We have presented a fixed-point abstract semantics that is parameterized by a domain of path descriptions and a domain of abstract substitutions. Two abstractions of paths are used to exemplify the usefulness of the abstract semantics in improving precision of an analysis. The abstract semantics can be used with abstract domains that have been developed without taking path information into account.

9. REFERENCES

[1] R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of computer programming*, 19(3):133–181, 1992.

[2] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

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

[4] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for prolog. *The ACM Transaction on Programming Languages and Systems*, 16(1):35–101, 1994.

[5] M. Codish, D. Dams, and E. Yardani. Bottom-up abstract interpretation of logic programs. *Journal of Theoretical Computer Science*, 124:93–125, 1994.

[6] M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. *Journal of Theoretical Computer Science*, 238:131–159, 2000.

[7] A. Cortesi, G. Filé, and W. Winsborough. Optimal groundness analysis using propositional logic. *Journal of Logic Programming*, 27(2):137–168, 1996.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL'77*, pages 238–252. The ACM Press, 1977.

[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] S.K. Debray. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, 1989.

[11] S.K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

[12] S.K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–230, 1988.

[13] P. Deransart, B. Lorho, and J. Małuszynski, editors. *Proceedings of the First International Workshop on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science 348. Springer, 1988.

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

[15] M. Hermenegildo, R. Warren, and S.K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(4):349–366, 1992.

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

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

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

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

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

[21] R. A. Kowalski and K. A. Bowen, editors. *Proceedings of the Fifth International Conference and Symposium on Logic Programming*. The MIT Press, 1988.

[22] G. Levi and F. Spoto. An Experiment in Domain Refinement: Type Domains and Type Representations for Logic Programs. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, Lecture Notes in Computer Science 1490, pages 152–169. Springer-Verlag, 1998.

[23] T. Lindgren. Control flow analysis of Prolog. In J.W. Lloyd, editor, *Logic Programming, Proceedings of the 1995 International Symposium*, pages 432–446. The MIT Press, 1995.

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

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

[26] L. Lu. A polymorphic type analysis in logic programs by abstract interpretation. *Journal of Logic Programming*, 36(1):1–54, 1998.

[27] L. Lu. A precise type analysis of logic programs. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of the Second International ACM SIGPLAN Conference on Principles and Practices of Declarative Programming*, pages 214–225. The ACM Press, 2000.

[28] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In Kowalski and Bowen [21], pages 733–748.

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

[30] K. Marriott, H. Søndergaard, and N. D. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[31] 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.

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

[33] F. Nielson and H. Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proc. POPL'97*, pages 332–345. ACM Press, 1997.

[34] U. Nilsson. Towards a framework for the abstract interpretation of logic programs. In Deransart et al. [13], pages 68–82.

[35] D. De Schreye and M. Bruynooghe. An application of abstract interpretation in source level program transformation. In Deransart et al. [13], pages 35–57.

[36] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis*, pages 189–233. Prentice Hall International, 1981.

[37] H. Søndergaard. An application of abstract interpretation of logic programs: occur check problem. In B. Robinet and R. Wilhelm, editors, *ESOP 86, European Symposium on Programming*, Lecture Notes in Computer Science 213, pages 324–338. Springer, 1986.

[38] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings of the Third International Conference on Logic Programming*, pages 84–98, London, U.K., 1986.

[39] A. Taylor. Removal of dereferencing and trailing in Prolog compilation. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 48–60. The MIT Press, 1989.

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

[41] K. Verschaetse and D. De Schreye. Deriving termination proofs for logic programs, using abstract procedures. In Furukawa [14], pages 301–315.

[42] A. Waern. An implementation technique for the abstract interpretation of Prolog. In Kowalski and Bowen [21], pages 700–710.

[43] W. Winsborough. Path-Dependent Reachability Analysis for Multiple Specialization. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 133–153, Cleveland, Ohio, USA, 1989.

APPENDIX
A. PROOFS
A.1 AUXILIARY LEMMAS

This section contains auxiliary lemmas used in proofs. Let $rang(\theta)$ be the range of a substitution θ. Let $o_1 \cong o_2$ denote the relation $o_1 = \rho(o_2)$ for some renaming substitution ρ. $\cong$ is an equivalence relation. We shall omit the parentheses in the application of a substitution to a term and the function composition operator $\circ$ in the composition of two substitutions when the omission does not incur any ambiguity.

LEMMA A.1. *Let ρ be a renaming such that $(vars(\rho(a)) \cup vars(\rho(\phi))) \cap (vars(b) \cup vars(\psi)) = \emptyset$. If $(\rho(\phi))(\rho(a))$ and $\psi(b)$ unify then $\rho(a)$ and b unify.*

PROOF. *Let a' be $\rho(a)$ and ϕ' be $\rho(\phi)$. If $\phi'(a')$ and $\psi(b)$ unify then there is a substitution θ such that $\theta(\phi'(a')) = \theta(\psi(b))$. We have $vars(a') \cap dom(\psi) = \emptyset$ and $rang(\psi) \cap dom(\phi') = \emptyset$ and $vars(b) \cap dom(\phi') = \emptyset$. Hence, $\theta\phi'\psi(a') = \theta\phi'\psi(b)$. Therefore, $\rho(a)$ and b unify.* □

LEMMA A.2. *Let A and B be two atoms, and ρ_1 and ρ_2 be two renamings such that*

$$dom(\rho_1) = dom(\rho_2) \supseteq vars(B) \tag{11}$$

$$rang(\rho_i) \cap vars(A) = \emptyset \quad for \quad i = 1, 2 \tag{12}$$

Then

(a) A and $\rho_1 B$ unify iff A and $\rho_2 B$ unify.
(b) $mgu(A, \rho_1 B) \uparrow vars(A) \cong mgu(A, \rho_2 B) \uparrow vars(A)$.
(c) $mgu(A, \rho_1 B)\rho_1 \uparrow dom(\rho_1) \cong mgu(A, \rho_2 B)\rho_2 \uparrow dom(\rho_2)$.

PROOF. *Let*

$$\begin{aligned} vars(A) &= \{X_1, \cdots, X_k\} \\ dom(\rho_1) &= dom(\rho_2) = \{V_1, \cdots, V_l\} \\ \rho_1 &= \{V_1/Y_1, \cdots, V_l/Y_l\} \\ \rho_2 &= \{V_1/Z_1, \cdots, V_l/Z_l\} \\ \rho_3 &= \{Z_1/Y_1, \cdots, Z_l/Y_l\} \\ \rho_4 &= \{Y_1/Z_1, \cdots, Y_l/Z_l\} \\ \mathcal{Y} &= \{Y_1, \cdots, Y_l\} \\ \mathcal{Z} &= \{Z_1, \cdots, Z_l\} \\ \mathcal{V} &= \{V_1, \cdots, V_l\} \end{aligned}$$

We have

$$\rho_1 = \rho_3\rho_2 \uparrow \mathcal{V} \tag{13}$$

$$\rho_2 = \rho_4\rho_1 \uparrow \mathcal{V} \tag{14}$$

Suppose that A and $\rho_1 B$ unify with the most general unifier

$$\theta_1 = \{X_{i_1}/x_{i_1}, \cdots, X_{i_s}/x_{i_s}, Y_{j_1}/y_{j_1}, \cdots, Y_{j_t}/y_{j_t}\} \tag{15}$$

with $1 \leq i_1 \leq \cdots \leq i_s \leq k$ *and* $1 \leq j_1 \leq \cdots \leq j_t \leq l$. *Define*

$$y_h \stackrel{def}{=} \begin{cases} Y_h & If\ h \notin \{j_1, j_2, \cdots, j_t\} \\ y_h & If\ h \in \{j_1, j_2, \cdots, j_t\} \end{cases} \quad (16)$$

By Eq. 15-16, we have

$$\theta_1\rho_1 \uparrow \mathcal{V} = \{V_1/y_1, \cdots, V_l/y_l\} \quad (17)$$

$\theta_1\rho_3 A = \theta_1 A = \theta_1\rho_1 B = \theta_1(\rho_3\rho_2 \uparrow \mathcal{V})B = \theta_1\rho_3\rho_2 B$ *by Eq. 11, 12, 13, 15 and 16. So,* A *and* $\rho_2 B$ *unify with* $\theta_1\rho_3$ *being one of their unifiers if* $\theta_1 = mgu(A, \rho_1 B)$.

Suppose A *and* $\rho_2 B$ *unify with most general unifier*

$$\theta_2 = \{X_{u_1}/\overline{x}_{u_1}, \cdots, X_{u_p}/\overline{x}_{u_p}, Z_{v_1}/z_{v_1}, \cdots, Z_{v_q}/z_{v_q}\} \quad (18)$$

with $1 \leq u_1 \leq \cdots \leq u_p \leq k$ *and* $1 \leq v_1 \leq \cdots \leq v_q \leq l$. *Define*

$$z_h \stackrel{def}{=} \begin{cases} Z_h & If\ h \notin \{v_1, v_2, \cdots, v_q\} \\ z_h & If\ h \in \{v_1, v_2, \cdots, v_q\} \end{cases} \quad (19)$$

By Eq. 18-19, we have

$$\theta_2\rho_2 \uparrow \mathcal{V} = \{V_1/z_1, \cdots, V_l/z_l\} \quad (20)$$

$\theta_2\rho_4 A = \theta_2 A = \theta_2\rho_2 B = \theta_2(\rho_4\rho_1 \uparrow \mathcal{V})B = \theta_2\rho_4\rho_1 B$ *by equations 11-12, 14, and 18-19. So,* A *and* $\rho_1 B$ *unify with* $\theta_2\rho_4$ *being one of their unifiers if* $\theta_2 = mgu(A, \rho_2 B)$. *Therefore, (a) holds.*

The following equations results from Eq. 15 and 18.

$$\theta_1\rho_3 = \left(\begin{array}{l} \{X_{i_1}/x_{i_1}, \cdots, X_{i_s}/x_{i_s}\} \\ \cup\ \{Y_{j_o}/y_{j_o} \mid 1 \leq o \leq t \wedge Y_{j_o} \notin \mathcal{Z}\} \\ \cup\ \{Z_1/y_1, \cdots, Z_l/y_l\} \end{array} \right) \quad (21)$$

$$\theta_2\rho_4 = \left(\begin{array}{l} \{X_{u_1}/\overline{x}_{u_1}, \cdots, X_{u_p}/\overline{x}_{u_p}\} \\ \cup\ \{Z_{v_o}/z_{v_o} \mid 1 \leq o \leq q \wedge Z_{v_o} \notin \mathcal{Y}\} \\ \cup\ \{Y_1/z_1, \cdots, Y_l/z_l\} \end{array} \right) \quad (22)$$

Since $\theta_2\rho_4$ *(*$\theta_1\rho_3$*) is a unifier of* A *and* $\rho_1 B$ *(*$\rho_2 B$*), there is a substitution* ζ_1 *(*ζ_2*) such that* $\theta_2\rho_4 = \zeta_1\theta_1$ *(*$\theta_1\rho_3 = \zeta_2\theta_2$*). By Eq. 15 and 22 (Eq. 18 and 21), we have*

$$\left(\begin{array}{l} \{X_{u_1}/\overline{x}_{u_1}, \cdots, X_{u_p}/\overline{x}_{u_p}\} \\ \cup\ \{Z_{v_o}/z_{v_o} \mid 1 \leq o \leq q \wedge Z_{v_o} \notin \mathcal{Y}\} \\ \cup\ \{Y_1/z_1, \cdots, Y_l/z_l\} \end{array} \right) = \quad (23)$$
$$(\{X_{i_1}/x_{i_1}, \cdots, X_{i_s}/x_{i_s}\} \cup \{Y_{j_1}/y_{j_1}, \cdots, Y_{j_t}/y_{j_t}\})\zeta_1$$

$$\left(\begin{array}{l} \{X_{i_1}/x_{i_1}, \cdots, X_{i_s}/x_{i_s}\} \\ \cup\ \{Y_{j_o}/y_{j_o} \mid 1 \leq o \leq t \wedge Y_{j_o} \notin \mathcal{Z}\} \\ \cup\ \{Z_1/y_1, \cdots, Z_l/y_l\} \end{array} \right) = \quad (24)$$
$$(\{X_{u_1}/\overline{x}_{u_1}, \cdots, X_{u_p}/\overline{x}_{u_p}\} \cup \{Z_{v_1}/z_{v_1}, \cdots, Z_{v_q}/z_{v_q}\})\zeta_2$$

By Eq. 23 and 24, $\{X_{i_1}, \cdots, X_{i_s}\} \subseteq \{X_{u_1}, \cdots, X_{u_p}\}$ *and* $\{X_{u_1}, \cdots, X_{u_p}\} \subseteq \{X_{i_1}, \cdots, X_{i_s}\}$. *So,* $\{X_{i_1}, \cdots, X_{i_s}\} = \{X_{u_1}, \cdots, X_{u_p}\}$. *We have* $s = p$ *and* $i_o = u_o$ *for* $1 \leq o \leq s$. *We also have* $x_{i_o} = \zeta_2(\overline{x}_{i_o})$ *and* $\overline{x}_{i_o} = \zeta_1(x_{i_o})$. *So,* $x_{i_o} \cong \overline{x}_{i_o}$ *for* $1 \leq o \leq s$. *Therefore, (b) holds.*

By Eq. 23,

$$\begin{array}{ll} Y_h/z_h \in \zeta_1 & If\ h \notin \{j_1, \cdots, j_t\} \\ z_h = \zeta_1(y_h) & If\ h \in \{j_1, \cdots, j_t\} \end{array}$$

By Eq. 16, $y_h = Y_h$ *for* $h \notin \{j_1, \cdots, j_t\}$ *and hence* $z_h = \zeta_1(y_h)$ *for* $h \notin \{j_1, \cdots, j_t\}$. *So, for all* $1 \leq h \leq l$,

$$z_h = \zeta_1(y_h) \quad (25)$$

It can be proved in a similar way from Eq. 24 and 19 that

$$y_h = \zeta_2(z_h) \quad (26)$$

By Eq. 17, 20, 25 and 26, $\theta_1\rho_1 \uparrow \mathcal{V} \cong \theta_2\rho_2 \uparrow \mathcal{V}$. *Therefore, (c) holds.* □

COROLLARY A.3. *Let* A *and* B *be two atoms and* ρ *be a renaming such that* $dom(\rho) \supseteq vars(B)$. *If* $vars(A) \cap vars(B) = \emptyset$ *and* $vars(A) \cap vars(\rho B) = \emptyset$ *then* A *and* B *unify iff* A *and* ρB *unify, and*

$$mgu(A, B) \uparrow vars(B) \cong (mgu(A, \rho B)\ \rho) \uparrow vars(B)$$

PROOF. *The proof results immediately from lemma A.1.(a) and (c) by letting* $\rho_2 = \rho$ *and* ρ_1 *be a renaming such that* $\rho_1 X = X$ *for each* $X \in vars(B)$. □

COROLLARY A.4. *Let* A *and* B *be two atoms,* ρ_a *and* ρ_b *be renamings. If*

$$\begin{array}{rcl} dom(\rho_a) & \supseteq & vars(A) \\ dom(\rho_b) & \supseteq & vars(B) \\ vars(\rho_a A) \cap vars(B) & = & \emptyset \\ vars(\rho_b B) \cap vars(A) & = & \emptyset \end{array}$$

then $\rho_a A$ *and* B *unify iff* A *and* $\rho_b B$ *unify, and*

$$(mgu(\rho_a A, B)\ \rho_a) \uparrow dom(\rho_a) \cong mgu(A, \rho_b B) \uparrow vars(A)$$

PROOF. *We only prove the if part since the only if part is dual. Let* ρ'_b *be a renaming such that* $dom(\rho'_b) = dom(\rho_b)$, $vars(\rho'_b B) \cap vars(A) = \emptyset$ *and* $vars(\rho_a A) \cap vars(\rho'_b B) = \emptyset$.

Suppose that A *and* $\rho_b B$ *unify. By lemma A.1.(a),* A *and* $\rho'_b B$ *unify, and* $mgu(A, \rho'_b B) \uparrow vars(A) \cong mgu(A, \rho_b B) \uparrow vars(A)$ *by lemma A.1.(b). By corollary A.3,* $\rho_a A$ *and* $\rho'_b B$ *unify, and*

$$mgu(\rho_a A, \rho'_b B)\ \rho_a \uparrow vars(A) \cong mgu(A, \rho'_b B) \uparrow vars(A)$$

So, $mgu(\rho_a A, \rho'_b B)\ \rho_a \uparrow vars(A) \cong mgu(A, \rho_b B) \uparrow vars(A)$. *By corollary A.3,* $\rho_a A$ *and* B *unify,*

$$mgu(\rho_a A, B) \uparrow vars(\rho_a A) \cong mgu(\rho_a A, \rho'_b B) \uparrow vars(\rho_a A)$$

that implies

$$mgu(\rho_a A, B)\ \rho_a \uparrow vars(A) \cong mgu(\rho_a A, \rho'_b B)\rho_a \uparrow vars(A)$$

So, $mgu(\rho_a A, B)\rho_a \uparrow vars(A) \cong mgu(A, \rho_b B) \uparrow vars(A)$. *It now suffices to prove*

$$mgu(\rho_a A, B)\ \rho_a \uparrow dom(\rho_a) \cong mgu(\rho_a A, B)\ \rho_a \uparrow vars(A)$$

Let $\rho_a^1 = \rho_a \uparrow vars(A)$ *and* $\rho_a^2 = \rho_a \uparrow (dom(\rho_a) - vars(A))$. *We have* $\rho_a = \rho_a^1 \cup \rho_a^2$,

$$\begin{aligned} &mgu(\rho_a A, B)\ \rho_a \uparrow dom(\rho_a) \\ &\quad = mgu((\rho_a^1 \cup \rho_a^2)A, B)\ (\rho_a^1 \cup \rho_a^2) \uparrow dom(\rho_a) \\ &\quad = mgu(\rho_a^1 A, B)\rho_a^1 \uparrow vars(A) \cup \rho_a^2 \end{aligned}$$

and

$$\begin{aligned} &mgu(\rho_a A, B)\rho_a \uparrow vars(A) \\ &\quad = mgu(A(\rho_a^1 \cup \rho_a^2), B)(\rho_a^1 \cup \rho_a^2) \uparrow vars(A) \\ &\quad = mgu(\rho_a^1 A, B)\rho_a^1 \uparrow vars(A) \end{aligned}$$

We also have $rang(mgu(\rho_a^1 A, B)\rho_a^1 \uparrow vars(A)) \cap dom(\rho_a^2) = \emptyset$ *and* $dom(\rho_a^2) \cap vars(A) = \emptyset$. *So,*

$$\begin{aligned} &(mgu(\rho_a A, B)\rho_a \uparrow vars(A))\rho_a^2 \\ &\quad = (mgu(\rho_a^1 A, B)\rho_a^1 \uparrow vars(A))\rho_a^2 \\ &\quad = mgu(\rho_a^1 A, B)\rho_a^1 \uparrow vars(A) \cup \rho_a^2 \\ &\quad = mgu(\rho_a A, B)\rho_a \uparrow dom(\rho_a) \end{aligned}$$

Therefore, $mgu(\rho_a A, B)\rho_a \uparrow dom(\rho_a) \cong mgu(\rho_a A, B)\rho_a \uparrow vars(A)$ *since* ρ_a^2 *is a renaming.* □

LEMMA A.5. *Let* θ_1 *and* θ_2 *be two substitutions and* $\mathcal{V}$ *a set of variables.*

$$\theta_2\theta_1 \uparrow \mathcal{V} = \theta_2(\theta_1 \uparrow \mathcal{V}) \uparrow \mathcal{V}$$

PROOF. *Let* $(X/t) \in \theta_2\theta_1 \uparrow \mathcal{V}$. *Then* $X \in \mathcal{V}$. *Either* $X \in dom(\theta_1)$ *or* $X \notin dom(\theta_1) \wedge X \in dom(\theta_2)$. *If* $X \in dom(\theta_1)$ *then there is* t_1 *such that* $((X/t_1) \in \theta_1 \wedge t = \theta_2(t_1))$. *Since* $X \in \mathcal{V}$, $(X/t_1) \in \theta_1 \uparrow \mathcal{V}$ *and hence* $X/\theta_2(t_1) = (X/t) \in \theta_2(\theta_1 \uparrow \mathcal{V}) \uparrow \mathcal{V}$. *Otherwise,* $X \in dom(\theta_2)$, $(X/t) \in \theta_2$ *and* $(X/t) \in \theta_2\ (\theta_1 \uparrow \mathcal{V}) \uparrow \mathcal{V}$.

Let $(X/t) \in \theta_2(\theta_1 \uparrow \mathcal{V}) \uparrow \mathcal{V}$. *Then* $X \in \mathcal{V}$. *Either* $X \in dom(\theta_1 \uparrow \mathcal{V})$ *or* $X \notin \theta_1 \uparrow \mathcal{V} \wedge X \in dom(\theta_2)$. *If* $X \in dom(\theta_1 \uparrow \mathcal{V})$ *then there is* t_2 *such that* $((X/t_2) \in \theta_1 \uparrow \mathcal{V} \wedge t = \theta_2(t_2))$. $(X/t_2) \in \theta_1$ *and* $(X/t) \in \theta_2\theta_1$. *So,* $(X/t) \in \theta_2\theta_1 \uparrow \mathcal{V}$. *Otherwise,* $(X/t) \in \theta_2$ *and* $X \notin dom(\theta_1) \cap \mathcal{V}$. *So,* $(X/t) \in \theta_2\theta_1 \uparrow \mathcal{V}$. □

A.2 PROOF OF LEMMA 3.1

The proof has two parts. The first part corresponds to procedure-entry and the second part to procedure-exit.

Consider procedure entry first. Let $\tau_q(\rho_C(A_q)G)$ be a goal in SLD where A_q is an atom in the body of a clause C and ρ_C the renaming substitution applied to C, $\mathcal{V}_C = vars(C)$ and $(q\delta', \sigma_q)S$ the current VSLD state. Let $C' = (H \leftarrow B)$ be an arbitrary clause with $p = entry(C')$ and $\mathcal{V}_{C'} = vars(C')$. We prove that if $\sigma_q \uparrow \mathcal{V}_C \cong \tau_q\rho_C \uparrow \mathcal{V}_C$ then $\tau_p(\rho_{C'}(B)\tau_q(G))$ is derived from $\tau_q(\rho_C(A_q)G)$ using clause C' iff $(q\delta', \sigma_q)S \overset{P}{\rightsquigarrow} (pq\delta', \sigma_p)(q\delta', \sigma_q)S$ and $\sigma_p \uparrow \mathcal{V}_{C'} \cong \tau_p\rho_{C'} \uparrow \mathcal{V}_{C'}$ where $\rho_{C'}$ is the renaming applied to C' in SLD.

Let $\sigma_q \uparrow \mathcal{V}_C \cong \tau_q\rho_C \uparrow \mathcal{V}_C$. Then there is a renaming ζ such that

$$\zeta(\sigma_q \uparrow \mathcal{V}_C) = \tau_q\rho_C \uparrow \mathcal{V}_C \tag{27}$$

We have that $\tau_p(\rho_{C'}(B)\tau_q(G))$ is derived from $\tau_q(\rho_C(A_q)G)$ using clause C' iff $(q\delta', \sigma_q)S \overset{P}{\rightsquigarrow} (pq\delta', \sigma_p)(q\delta', \sigma_q)S$ by corollary A.4. Suppose that $\tau_p(\rho_{C'}(B)\tau_q(G))$ is derived from $\tau_q(\rho_C(A_q)G)$ using clause C'. Then

$$\begin{aligned} &\tau_q\rho_C A_q \\ &\quad = (\tau_q\rho_C \uparrow \mathcal{V}_C)A_q && (\because vars(A_q) \subseteq \mathcal{V}_C) \\ &\quad = (\zeta(\sigma_q \uparrow \mathcal{V}_C))A_q && (\because Eq.\ 27) \\ &\quad = \zeta\sigma_q A_q && (\because vars(A_q) \subseteq \mathcal{V}_C) \end{aligned} \tag{28}$$

and

$$\begin{aligned} &\tau_p\rho_{C'} \uparrow \mathcal{V}_{C'} \\ &\quad = mgu(\rho_{C'}H, \tau_q\rho_C A_q)\ \rho_{C'} \uparrow \mathcal{V}_{C'} \\ &\quad = mgu(\rho_{C'}H, \zeta\sigma_q A_q)\ \rho_{C'} \uparrow \mathcal{V}_{C'} && (\because Eq.\ 28) \end{aligned} \tag{29}$$

Let $\overline{\zeta}$ be the inverse of ζ and ψ be a renaming.

$$\begin{aligned} &\sigma_p \uparrow \mathcal{V}_{C'} \\ &\quad = mgu(H, \psi\sigma_q A_q) \uparrow \mathcal{V}_{C'} \\ &\quad = mgu(H, \psi\overline{\zeta}\zeta\sigma_q A_q) \uparrow \mathcal{V}_{C'} && (\because \overline{\zeta}\zeta\ is\ identity) \\ &\quad = mgu(H, (\psi\overline{\zeta})(\zeta\sigma_q A_q)) \uparrow \mathcal{V}_{C'} \\ &\quad = mgu(H, (\psi\overline{\zeta})(\zeta\sigma_q A_q)) \uparrow vars(H) \end{aligned} \tag{30}$$

$\sigma_p \uparrow \mathcal{V}_{C'} \cong \tau_p\rho_{C'} \uparrow \mathcal{V}_{C'}$ by corollary A.4 and Eq. 29-30. This completes the first part of the proof.

We now consider procedure exit. Let $r = exit(C')$, the current VSLD state be $(r\delta''pq\delta', \sigma_r)(q\delta', \sigma_q)S$ and the current goal in SLD be $\tau_r(G)$. Let $(r\delta''pq\delta', \sigma_r)(q\delta', \sigma_q)S \overset{P}{\rightsquigarrow} (q^+r\delta''pq\delta', \sigma_{q^+})S$. We prove that if $\sigma_r \uparrow \mathcal{V}_{C'} \cong \tau_r\rho_{C'} \uparrow \mathcal{V}_{C'}$ then $\sigma_{q^+} \uparrow \mathcal{V}_C \cong \tau_r\rho_C \uparrow \mathcal{V}_C$. Let ζ' be a renaming such that $\sigma_r \uparrow \mathcal{V}_{C'} = \zeta'(\tau_r\rho_{C'} \uparrow \mathcal{V}_{C'})$ and $\overline{\zeta'}$ be the inverse of ζ'. $\sigma_r \uparrow \mathcal{V}_{C'} = \zeta'\tau_r\rho_{C'} \uparrow \mathcal{V}_{C'}$. Let ϕ' be a renaming and θ be the computed answer to $\tau_p(\rho_{C'}(B))$. We have, $\tau_r = \theta\tau_p$ and

$$\begin{aligned} &\phi'\sigma_r H \\ &\quad = \phi'\zeta'\tau_r\rho_{C'}H \\ &\quad = \phi'\zeta'\theta\eta\tau_q\rho_{C'}H \\ &\quad = \phi'\zeta'\theta\eta\rho_{C'}H && (\because vars(\rho_{C'}C') \cap vars(\rho_C C) = \emptyset) \\ &\quad = \phi'\zeta'\theta\eta\tau_q\rho_C A_q \end{aligned} \tag{31}$$

where $\eta = mgu(\rho_{C'}H, \tau_q\rho_C A_q)$. By Eq. 27,

$$\sigma_q \uparrow \mathcal{V}_C = \overline{\zeta}\tau_q\rho_C \uparrow \mathcal{V}_C \tag{32}$$

So,

$$\begin{aligned} &\sigma_q A_q \\ &\quad = (\overline{\zeta}\tau_q\rho_C \uparrow \mathcal{V}_C)A_q && (\because Eq.\ 32) \\ &\quad = \overline{\zeta}\tau_q\rho_C A_q && (\because vars(A_q) \subseteq \mathcal{V}_C) \end{aligned} \tag{33}$$

Therefore, letting $A = \tau_q\rho_C A_q$,

$$\begin{aligned} &\sigma_{q^+} \uparrow \mathcal{V}_C \\ &\quad = mgu(\overline{\zeta}A, \phi'\zeta'\theta\eta A)\ \overline{\zeta}\tau_q\rho_C \uparrow \mathcal{V}_C \\ &\quad = (mgu(\overline{\zeta}A, \phi'\zeta'\theta\eta A)\overline{\zeta} \uparrow vars(A))\tau_q\rho_C \uparrow \mathcal{V}_C && (\because A.5) \\ &\quad \cong (mgu(A, \phi'\zeta'\theta\eta A) \uparrow vars(A))\tau_q\rho_C \uparrow \mathcal{V}_C && (\because A.3) \\ &\quad = mgu(A, \phi'\zeta'\theta\eta A)\tau_q\rho_C \uparrow \mathcal{V}_C && (\because A.5) \\ &\quad = \phi'\zeta'\theta\eta\tau_q\rho_C \uparrow \mathcal{V}_C \\ &\quad \cong \theta\eta\tau_q\rho_C \uparrow \mathcal{V}_C \\ &\quad = \tau_r\rho_C \uparrow \mathcal{V}_C \end{aligned}$$

A.3 PROOF OF LEMMA 4.1

We first characterize $[P]$ as the fixed-point of the following function.

$$F_P(X) \stackrel{def}{=} \bigcup_{0\leq \jmath\leq 2} F_P^{\jmath}(X) \tag{34}$$

$$F_P^0(X) \stackrel{def}{=} \{(p,\theta)\$ \mid p \in \mathcal{N}^0 \wedge \theta \in \Theta_p\} \tag{35}$$

$$F_P^1(X) \stackrel{def}{=} \left\{ (pq\delta', \sigma)(q\delta', \theta)S \;\middle|\; \begin{array}{c}(p,q) \in \mathcal{E}^1 \wedge (q\delta', \theta)S \in X \\ \wedge \\ \sigma = uf(A_q, \theta, H_p, \epsilon) \neq \mathsf{fail}\end{array} \right\} \tag{36}$$

$$F_P^2(X) \stackrel{def}{=} \left\{ (pq\delta' p^{\text{-}}\delta'', \sigma)S \;\middle|\; \begin{array}{c}(p,q) \in \mathcal{E}^2 \\ \wedge \\ (q\delta' p^{\text{-}}\delta'', \theta)(p^{\text{-}}\delta'', \omega)S \in X \\ \wedge \\ \sigma = uf(H_q, \theta, A_{p^{\text{-}}}, \omega) \neq \mathsf{fail}\end{array} \right\} \tag{37}$$

The domain $\mathcal{D}$ of F_P is $\wp(\mathcal{S})$. $<\mathcal{D}, \subseteq>$ is a complete lattice and F_P is monotonic on $<\mathcal{D}, \subseteq>$. It is easy to see $[P] = lfpF_P$.

It is now sufficient to prove that $F_P \uparrow k \subseteq \gamma^\sharp(F_P^\sharp \uparrow k)$ for any ordinal k. The proof is done by transfinite induction.

Basis. $F_P \uparrow 0 = \emptyset = \gamma^\sharp(\bot^\sharp) = \gamma^\sharp(F_P^\sharp \uparrow 0)$.

Induction. Let $F_P \uparrow k' \subseteq \gamma^\sharp(F_P^\sharp \uparrow k')$ for any $k' < k$. If k is a limit ordinal then $F_P^\sharp \uparrow k = \sqcup^\sharp\{F_P^\sharp \uparrow k' \mid k' < k\}$. Therefore, $\gamma^\sharp(F_P^\sharp \uparrow k) \supseteq \gamma^\sharp(F_P^\sharp \uparrow k')$ for any $k' < k$ by Eq. 2. By the induction hypothesis, $\gamma^\sharp(F_P^\sharp \uparrow k) \supseteq F_P \uparrow k'$ for any $k' < k$. So, $F_P \uparrow k \subseteq \gamma^\sharp(F_P^\sharp \uparrow k)$.

Let k not be a limit ordinal. Let $S' \in F_P \uparrow k$. There is $0 \leq \jmath \leq 2$ such that $S' \in F_P^\jmath(F_P \uparrow (k-1))$ by Eq. 34 and Eq. 2.

Let $\jmath = 0$. By Eq. 35, $S' = (p,\theta)\$$ and $\theta \in \Theta_p$. So, by Eq. 3 and Eq. 2, $S' \in \gamma^\sharp(F_P^\sharp \uparrow k)$.

Let $\jmath = 1$. By Eq. 36, $S' = (pq\delta', \sigma)(q\delta', \theta)S$ such that $(p,q) \in \mathcal{E}^1$, $(q\delta', \theta)S \in F_P \uparrow (k-1)$ and $\sigma = uf(A_q, \theta, H_p, \epsilon) \neq \mathsf{fail}$. We have $(q\delta', \theta)S \in \gamma^\sharp(F_P^\sharp \uparrow (k-1))$ by the induction hypothesis. $S' \in \gamma^\sharp(F_P^\sharp \uparrow k)$ by Eq. 4 and Eq. 2 and the monotonicity of $F_P^\sharp$.

Let $\jmath = 2$. By Eq. 37, $S' = (pq\delta' p^{\text{-}}\delta'', \sigma)S$ and there is an ω such that $(p,q) \in \mathcal{E}^2$,

$$(q\delta' p^{\text{-}}\delta'', \theta)(p^{\text{-}}\delta'', \omega)S \in F_P \uparrow (k-1)$$

$$\sigma = uf(H_q, \theta, A_{p^{\text{-}}}, \omega) \neq \mathsf{fail}$$

$(q\delta' p^{\text{-}}\delta'', \theta)(p^{\text{-}}\delta'', \omega)S \in \gamma^\sharp(F_P^\sharp \uparrow (k-1))$ by the induction hypothesis. Therefore, $S' \in \gamma^\sharp(F_P^\sharp \uparrow k)$ by Eq. 5 and Eq. 2 and the monotonicity of $F_P^\sharp$.

Therefore, $F_P \uparrow k \subseteq \gamma^\sharp(F_P^\sharp \uparrow k)$ for any ordinal k.

A.4 PROOF OF THEOREM 5.1

(C4) implies that $F_P^\flat$ is monotonic and therefore $lfpF_P^\flat$ exists. It suffices to prove that, for any $X^\flat \in \mathcal{D}^\flat$, $F_P^\sharp \circ \gamma^\flat(X^\flat) \sqsubseteq^\sharp \gamma^\flat \circ F_P^\flat(X^\flat)$.

Let $\sigma \in [F_P^\sharp \circ \gamma^\flat(X^\flat)](p)(\delta)$. We need to prove

$$\sigma \in [\gamma^\flat \circ F_P^\flat(X^\flat)](p)(\delta)$$

We have $p \in \mathcal{N}^\jmath$ for some $0 \leq \jmath \leq 2$.

Let $\jmath = 0$. By Eq. 3, $\sigma \in \gamma_{\mathcal{V}_p}(\theta_p^\flat)$ and $\delta = p$. By Eq. 8, $\sigma \in \gamma_{\mathcal{V}_p}([F_P^\flat(X^\flat)](p)(\beta(\delta)))$. Thus, $\sigma \in [\gamma^\flat \circ F_P^\flat(X^\flat)](p)(\delta)$ by Eq. 7.

Let $\jmath = 1$. By Eq. 4, there is $q \in \mathcal{N}$ and $\delta' \in \Delta$ such that $\delta = pq\delta'$ and $\sigma \in uf^\sharp(A_q, [\gamma^\flat(X^\flat)](q)(q\delta'), H_p, \{\epsilon\})$. By Eq. 4 and Eq. 7, C3 and the monotonicity of function $uf^\sharp$ in its fourth argument,

$$\begin{array}{rcl}\sigma & \in & uf^\sharp(A_q, \gamma_{\mathcal{V}_q}(X^\flat(q)(\beta(q\delta'))), H_p, \gamma_{\mathcal{V}_p}(\epsilon_{\mathcal{V}_p})) \\ & \subseteq & \gamma_{\mathcal{V}_p} \circ uf^\flat_{\mathcal{V}_q,\mathcal{V}_p}(A_q, X^\flat(q)(\beta(q\delta')), H_p, \epsilon_{\mathcal{V}_p})\end{array}$$

So, by Eq. 7 and Eq. 9 and the monotonicity of $\gamma_{\mathcal{V}_p}$,

$$\begin{array}{rcl}\sigma & \in & \gamma_{\mathcal{V}_p}([F_P^\flat(X^\flat)](p)(\beta(pq\delta')) \\ & \subseteq & \gamma_{\mathcal{V}_p}([F_P^\flat(X^\flat)](p)(\beta(pq\delta')) \\ & = & [\gamma^\flat \circ F_P^\flat(X^\flat)](p)(pq\delta') \\ & = & [\gamma^\flat \circ F_P^\flat(X^\flat)](p)(\delta)\end{array}$$

Let $\jmath = 2$. There are $q \in \mathcal{N}$ and $\delta', \delta'' \in \Delta$ such that $\delta = pq\delta' p^{\text{-}}\delta''$ and

$$\sigma \in uf^\sharp(H_q, [\gamma^\flat(X^\flat)](q)(q\delta' p^{\text{-}}\delta''), A_{p^{\text{-}}}, [\gamma^\flat(X^\flat)](p^{\text{-}})(p^{\text{-}}\delta''))$$

by Eq. 5. We have $\beta(p^{\text{-}}\delta'') \ll \beta(q\delta' p^{\text{-}}\delta'')$. By Eq. 7 and Eq. 10,

$$\begin{array}{rcl}\sigma & \in & uf^\sharp(H_q, \gamma_{\mathcal{V}_q}(X^\flat(q)(\beta(q\delta' p^{\text{-}}\delta''))), \\ & & \qquad A_{p^{\text{-}}}, \gamma_{\mathcal{V}_{p^{\text{-}}}}(X^\flat(p^{\text{-}})(\beta(p^{\text{-}}\delta'')))) \\ & \subseteq & \gamma_{\mathcal{V}_{p^{\text{-}}}} \circ uf^\flat_{\mathcal{V}_q,\mathcal{V}_p}(H_q, X^\flat(q)(\beta(q\delta' p^{\text{-}}\delta'')), \\ & & \qquad A_{p^{\text{-}}}, X^\flat(p^{\text{-}})(\beta(p^{\text{-}}\delta''))) \\ & = & \gamma_{\mathcal{V}_p} \circ uf^\flat_{\mathcal{V}_q,\mathcal{V}_p}(H_q, X^\flat(q)(\beta(q\delta' p^{\text{-}}\delta'')), \\ & & \qquad A_{p^{\text{-}}}, X^\flat(p^{\text{-}})(\beta(p^{\text{-}}\delta''))) \\ & = & \gamma_{\mathcal{V}_p}([F_P^\flat(X^\flat)](p)(\beta(\delta))) \\ & = & [\gamma^\flat \circ F_P^\flat(X^\flat)](p)(\beta(\delta))\end{array}$$

since $\mathcal{V}_p = \mathcal{V}_{p^{\text{-}}}$