Path Dependent Analysis of Logic Programs

Lunjin Lu (lunjin@acm.org) *
Oakland University, Rochester, MI 48309, USA.

Abstract. This paper presents an abstract semantics that uses information about execution paths to improve precision of data flow analyses of logic programs. The abstract semantics is illustrated 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

Semantic-based program analysis provides useful information about run-time properties of programs to compilers, debuggers and other program manipulation tools. It has been instrumental in bringing efficiency to implementations of programming languages. Context sensitivity is one of primary methods for obtaining precise interprocedural data flow analysis. Context sensitive analysis records a piece of data flow information at each program point of a procedure for each context in which the procedure is called. In other words, at each program point, multiple pieces of data flow information are recorded and each is tagged with a context. There are two popular approaches to context sensitive analysis. In the assumption set approach, context information is represented as a set of descriptions of memory stores in which a procedure is called. In the call string approach, context information is represented as a sequence of call sites (program points) of uncompleted procedure calls.

In the call string approach that originated from Sharir and Pnueli[50], each piece of data flow information is tagged with a call string that records the history of uncompleted procedure calls along which that information is propagated. The call string on a piece of information is updated whenever a propagation step associated with a call statement or return statement is performed. The call string approach has since become one of the most popular mechanisms in obtaining context-sensitivity in interprocedural analysis, see references in [48, 46].

* The work is supported in part by the National Science Foundation (CCR-0131862). A preliminary version of this paper appeared in Proceedings of PEPM '02, Jan. 14-15, 2002 Portland, OR, USA

Almost all analyses for logic programs are formulated as abstract interpretations. In abstract interpretation [10], program analyses are viewed as program executions over non-standard data domains. Design of an analysis usually begins with defining a collecting semantics that associates with each program point a set of memory stores that are obtained at the program point. The analysis is then defined as an approximation of the collecting semantics. The approximation 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). Much research has been done in abstract interpretation of logic programs [11]. A number of (generic) abstract semantics, also called frameworks or schemes [4, 6, 17, 23, 26, 38, 39, 43] have been proposed for analysis of logic programs; and they have been specialised for the detection of determinacy [13], data dependency analyses [14, 21], mode inference [14, 53], program transformation [49], type inference [22, 33, 34], termination proof [55, 5], etc.

In logic programs, interprocedural control flow is maximal whilst intraprocedural control flow is minimal. It is therefore natural to use context information to improve precision of logic program analysis. However, context sensitivity of logic program analysis has not attracted much attention and it has been a by-product of approximating top-down evaluation mechanisms. In particular, information about the context of a call has been exclusively captured by recording information about the memory stores 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 parameterised by both an abstraction of execution paths and an abstraction of memory stores. Furthermore, the abstract semantics allows path abstractions that are not call strings.

In addition to ensuring that procedure returns match corresponding procedure calls, call strings provide additional information that is not present in assumption sets: call sites. This information is useful in programming tools such as abstract diagnosers [9] that must trace and locate the source of discrepancies between derived and intended properties. See section 6.2 for an example. A distinguishing characteristics of logic programs is non-determinism. A procedure in a logic program is usually defined by several clauses and the clause invoked to execute a call is non-deterministically chosen at run-time. This is in contrast to an imperative program in which a procedure has only one definition. Thus, the call string approach need be adapted in order to differentiate data flow information produced by different clauses that define the same procedure. This requires keeping track of information about

completed as well as uncompleted calls. This is achieved by formulating a collecting semantics that is parameterised by execution paths. The collecting semantics is derived from an operational semantics that is the result of instrumenting the LD resolution (SLD resolution with the left-to-right computation rule) with execution paths and explicit memory stores. The collecting semantics is then approximated by applying independent abstractions of execution paths and memory stores. The result is a system of data flow equations whose least solution associates each description of execution paths with a description of data. The system of data flow equations can then be solved using a least fixed point algorithm such as work-list and round-robin algorithms.

The remainder of this paper is organised as follows. Section 2 briefly recalls on basic concepts in abstract interpretation and logic programming, and introduces notations used later in this paper. Section 3 reformulates LD resolution in order to facilitate the derivation of a collecting semantics. Section 4 derives the collecting semantics from the operational semantics. Section 5 derives the abstract semantics from the collecting semantics, and gives sufficient conditions for the abstract semantics to approximate safely the collecting semantics. In section 6, we show how the abstract semantics can be specialised by two examples. The first example uses call strings 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 generalised to analyse logic programs with negation and builtin predicates as in [4]. Proofs are included in an appendix.

2. Preliminaries

This section first recalls some basic concepts in abstract interpretation and logic programming and then formalises the notion of an execution path. The reader is referred to [30] and [10] for more detailed exposition on logic programming and abstract interpretation respectively.

We begin with some preliminary notations. Let Φ be a set of symbols. The set of all strings over Φ is denoted Φ^* and the set of all non-empty strings over Φ is denoted $\Phi^\dagger$. Let Λ be the empty string. Then $\Phi^\dagger = \Phi^* \backslash \{\Lambda\}$. The concatenation of two strings x and y is denoted by their juxtaposition xy. Let X and Y be two sets of strings. The juxtaposition XY of X and Y is defined $XY \stackrel{def}{=} \{xy \mid x \in X \wedge y \in Y\}$.

Church's lambda notation is sometimes used so that a function $f(x) = e(x)$ will be denoted by $\lambda x.e(x)$. The function composition operator $\circ$ is defined as $f \circ g \stackrel{def}{=} \lambda x.f(g(x))$.

2.1. Abstract Interpretation

A semantics of a program is given by an interpretation $\langle (C, \sqsubseteq_C), \mathcal{C} \rangle$ where $(C, \sqsubseteq_C)$ is a complete lattice and $\mathcal{C}$ is a monotone function on $(C, \sqsubseteq_C)$. The semantics is defined as the least fixed point *lfp* $\mathcal{C}$ of $\mathcal{C}$. The concrete semantics of the program is given by the concrete interpretation $\langle (C, \sqsubseteq_C), \mathcal{C} \rangle$ while an abstract semantics is given by an abstract interpretation $\langle (A, \sqsubseteq_A), \mathcal{A} \rangle$. The correspondence between the concrete and the abstract domains is formalised by a Galois connection (α, γ) between $(C, \sqsubseteq_C)$ and $(A, \sqsubseteq_A)$. A Galois connection between A and C is a pair of monotone functions $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ satisfying $\forall c \in C.(c \sqsubseteq_C \gamma \circ \alpha(c))$ and $\forall a \in A.(\alpha \circ \gamma(a) \sqsubseteq_A a)$. The function α is called an abstraction function and the function γ a concretisation function. A sufficient condition for *lfp* $\mathcal{A}$ to be a safe abstraction of *lfp* $\mathcal{C}$ is $\forall a \in A.(\alpha \circ \mathcal{C} \circ \gamma(a) \sqsubseteq_A \mathcal{A}(a))$ or equivalently $\forall a \in A.(\mathcal{C} \circ \gamma(a) \sqsubseteq_C \gamma \circ \mathcal{A}(a))$, according to propositions 24 and 25 in [11]. The abstraction and concretisation functions in a Galois connection uniquely determine each other; and a complete meet-morphism $\gamma : A \mapsto C$ induces a Galois connection (α, γ) with $\alpha(c) = \sqcap_A \{a \mid c \sqsubseteq_C \gamma(a)\}$. A function $\gamma : A \mapsto C$ is a complete meet-morphism iff $\gamma(\sqcap_A X) = \sqcap_C \{\gamma(x) \in X\}$ for any $X \subseteq A$. Thus, an analysis can be formalised as a tuple $(\langle (C, \sqsubseteq_C), \mathcal{C} \rangle, \gamma, \langle (A, \sqsubseteq_A), \mathcal{A} \rangle)$ such that $\langle (C, \sqsubseteq_C), \mathcal{C} \rangle$ and $\langle (A, \sqsubseteq_A), \mathcal{A} \rangle$ are interpretations, γ is a complete meet-morphism from $(C, \sqsubseteq_C)$ to $(A, \sqsubseteq_A)$, and $\forall a \in A.(\mathcal{C} \circ \gamma(a) \sqsubseteq_C \gamma \circ \mathcal{A}(a))$.

2.2. Logic Programs

Let Σ be a set of function symbols, Π a set of predicate symbols and Vars a denumerable set of variables. Each function or predicate symbol has an arity which is a non-negative integer. We write $f/n \in \Sigma$ for an n-ary function symbol f in Σ and $p/n \in \Pi$ for an n-ary predicate symbol p in Π. The set of all terms, denoted Term, is the smallest set satisfying: (i) $\mathsf{Vars} \subseteq \mathsf{Term}$; and (ii) if $\{t_1, \cdots, t_n\} \subseteq \mathsf{Term}$ and $f/n \in \Sigma$ then $f(t_1, \cdots, t_n) \in \mathsf{Term}$. The set of all atoms that can be constructed from Π and Term is denoted Atom; $\mathsf{Atom} = \{p(t_1, \cdots, t_n) \mid (p/n \in \Pi) \wedge (\{t_1, \cdots, t_n\} \subseteq \mathsf{Term})\}$.

A clause C is a formula of the form $H \leftarrow A_1 A_2 \cdots A_n \blacksquare$ where $H \in \mathsf{Atom} \cup \{\square\}$ 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. Note that $\square$ denotes the empty head and $\blacksquare$ denotes the empty body. A query is a clause whose head is $\square$. A program is a set of clauses of which one is a query. The query initiates the execution of the program. A goal is a sequence

of atoms interspersed with occurrences of ■ and is always terminated with an ■. The set of all goals is $\mathsf{Goal} \stackrel{def}{=} (\mathsf{Atom}^*\{■\})^\dagger$.

Memory stores that exist during the execution of a logic program are called substitutions. A substitution θ is a mapping from Vars to Term such that $dom(\theta) \stackrel{def}{=} \{x \mid (x \in \mathsf{Vars}) \wedge (\theta(x) \neq x)\}$ is finite. The set $dom(\theta)$ is called the domain of θ. Let $dom(\theta) = \{x_1, \cdots, x_n\}$. Then θ is written as $\{x_1/\theta(x_1), \cdots, x_n/\theta(x_n)\}$. A substitution θ is idempotent if $\theta \circ \theta = \theta$. The set of idempotent substitutions is denoted Sub; and the identity substitution is denoted ϵ. Let $\mathsf{Sub}_{fail} \stackrel{def}{=} \mathsf{Sub} \cup \{fail\}$ and extend $\circ$ by $\theta \circ fail \stackrel{def}{=} fail$ and $fail \circ \theta \stackrel{def}{=} fail$ for any $\theta \in \mathsf{Sub}_{fail}$. The restriction of a substitution θ to a set V of variables is defined as

$$\theta \uparrow V \stackrel{def}{=} \lambda x.(\text{if } x \in V \text{ then } \theta(x) \text{ else } x)$$

Substitutions are not distinguished from their homomorphic extensions to various syntactic categories.

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 Eqn. For a set E of equations, $mgu : \wp(\mathsf{Eqn}) \mapsto \mathsf{Sub}_{fail}$ returns either a most general unifier for E if E is unifiable or $fail$ otherwise. Let $mgu(l, r)$ stand for $mgu(\{l = r\})$.

The set of variables in a syntactic object o is denoted $vars(o)$. A renaming substitution ρ is a substitution such that $\{\rho(x) \mid x \in \mathsf{Vars}\}$ is a permutation of Vars. The set of all renaming substitutions is denoted Ren. Define $ren(o_1, o_2) \stackrel{def}{=} \{\rho \in \mathsf{Ren} \mid vars(\rho(o_1)) \cap vars(o_2) = \emptyset\}$.

2.3. Program Graph

The inputs to an analysis are a program P and a description of a set Θ_ι of substitutions for the query $(\square \leftarrow Q)$ in P. For each $\theta \in \Theta_\iota$, $(\square \leftarrow \theta(Q))$ is a possible query to P. A clause C with body $A_1 A_2 \cdots A_n$■ is designated with $n+1$ different (program) points with j^{th} point immediately before A_j for $1 \leq j \leq n$ and $(n+1)^{th}$ immediately before ■. The leftmost point, denoted $entry(C)$, is called the entry point of C and the rightmost point, denoted $exit(C)$, is called the exit point of C. Let $\mathcal{N}_C$ denote the set of the points associated with C and $\mathcal{N}$ the set of all the points designated with clauses in P. Note that $\mathcal{N} = \bigcup\{\mathcal{N}_C \mid C \in P\}$.

Let $p \in \mathcal{N}$. We use p^- to denote the point to the left of p if p^- exists and p^+ to denote the point to the right of p if p^+ exists. Let $\mathbb{A}(p)$ denote the atom or the nullary symbol ■ to the right of p and $\mathbb{H}(p)$ denote the head of the clause with which p is associated. Note that if p and q are two program points in the same clause then $\mathbb{H}(p) = \mathbb{H}(q)$.

Let $p, q \in \mathcal{N}$, and q be the most recent program point that LD resolution has reached. There are two possibilities that LD resolution will reach p next. If q is the exit point of a clause C then LD resolution can reach p immediately only if C has been used to resolve with $\mathbb{A}(p^{\text{-}})$. If q is not an exit point then LD resolution can reach p only if p is the entry point of a clause that can be used to resolve with $\mathbb{A}(q)$. Note that if q is the exit point of the query then LD resolution has succeeded and will not visit any more program points. A graph $< \mathcal{N}, \mathcal{E} >$, called program graph, is used to represent the relation that "LD resolution will possibly visit p immediately after it visits q".

$$\mathcal{E} \stackrel{def}{=} \mathcal{E}^{call} \cup \mathcal{E}^{ret}$$

$$\mathcal{E}^{call} \stackrel{def}{=} \left\{ entry(C) \leftharpoonup q \;\middle|\; \begin{array}{c} q \in \mathcal{N} \\ \wedge \\ C \in P \\ \wedge \\ \rho \in ren(\mathbb{A}(q), \mathbb{H}(entry(C))) \\ \wedge \\ mgu(\rho(\mathbb{A}(q)), \mathbb{H}(entry(C))) \neq fail \end{array} \right\}$$

$$\mathcal{E}^{ret} \stackrel{def}{=} \{p \leftharpoonup exit(C) \mid entry(C) \leftharpoonup p^{\text{-}} \in \mathcal{E}^{call}\}$$

Edges in $\mathcal{E}^{call}$ correspond to procedure-entry operations, and edges in $\mathcal{E}^{ret}$ to procedure-exit operations. Note that $\mathcal{E}^{call} \cap \mathcal{E}^{ret} = \emptyset$ and $p \leftharpoonup q$ is an edge from q to p. Let ι be the entry point of the query. It is be the initial program point. Let $\mathcal{N}^{call} \stackrel{def}{=} \{entry(C) \mid C \in P\} \setminus \{\iota\}$ and $\mathcal{N}^{ret} \stackrel{def}{=} (\mathcal{N} \setminus \mathcal{N}^{call}) \setminus \{\iota\}$. A point in $\mathcal{N}^{call}$ is reached after performing a procedure-entry operation and a point in $\mathcal{N}^{ret}$ is reached after performing a procedure-exit operation. There is no edge to ι since the head $\square$ of the query $(\square \leftarrow Q)$ does not occur in the body of any clause in the program P.

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

$C_1 \equiv both(X, L, K) \leftarrow \textcircled{1} member(X, L)\ \textcircled{2} member(X, K)\ \textcircled{3}$ ■
$C_2 \equiv member(X, [X|L]) \leftarrow \textcircled{4}$ ■
$C_3 \equiv member(X, [H|L]) \leftarrow \textcircled{5} member(X, L)\ \textcircled{6}$ ■
$Q \equiv \square \leftarrow \textcircled{7} both(X, L1, L2)\ \textcircled{8}$ ■

Let Θ_7 *be the set of substitutions* θ *such that* $\theta(X)$ *is a variable, and both* $\theta(L)$ *and* $\theta(K)$ *are ground terms. The set* $\mathcal{N}$ *contains 8*

program points with $\iota = 7$*, and* $\mathbb{A}(1) = member(X, L)$ *and* $\mathbb{A}(2) = member(X, K)$*. The edge* $5 \leftarrow 1$ *is in* $\mathcal{E}^{call}$ *and* $2 \leftarrow 6$ *in* $\mathcal{E}^{ret}$*. The program graph for the program is illustrated in figure 1.*

Figure 1. Program Graph for Example 2.1

∎

2.4. Execution paths

An execution path is a sequence $p_n p_{n-1} \cdots p_1$ of program points that are visited during an execution of the program. Note that the execution path starts with $p_1 = \iota$ and $p_i \leftarrow p_{i-1} \in \mathcal{E}$ for $2 \leq i \leq n$. Not every sequence $p_n p_{n-1} \cdots p_1$ of program points such that $p_i \leftarrow p_{i-1} \in \mathcal{E}$ for $2 \leq i \leq n$ is an execution path. For instance, 5 2 4 5 1 7 is not an execution path for the program in example 2.1 since it does not have proper nesting of procedure-entry and procedure-exit operations. We follow [12, 19, 20, 42, 44, 45, 46, 47] in using a formal grammar to model valid execution paths. The grammar is generated from an interprocedural flow defined in the following to ensure proper nesting.

$$IF \stackrel{def}{=} \left\{ (p^+ \leftarrow exit(C); entry(C) \leftarrow p) \;\middle|\; \begin{array}{c} p \in \mathcal{N} \\ \wedge \\ C \in P \\ \wedge \\ entry(C) \leftarrow p \in \mathcal{E}^{call} \end{array} \right\}$$

EXAMPLE 2.2. *Continue with example 2.1. The interprocedural flow for the program is*

$$IF = \left\{ \begin{array}{l} (8 \leftarrow 3; 1 \leftarrow 7) \;,\; (2 \leftarrow 4; 4 \leftarrow 1) \;,\; (2 \leftarrow 6; 5 \leftarrow 1) \;,\; (3 \leftarrow 4; 4 \leftarrow 2) \\ (3 \leftarrow 6; 5 \leftarrow 2) \;,\; (6 \leftarrow 4; 4 \leftarrow 5) \;,\; (6 \leftarrow 6; 5 \leftarrow 5) \end{array} \right\}$$

The grammar is derived from the interprocedural flow IF as follows.

$$\begin{array}{rcll} VP & \longrightarrow & VP_{p,\iota} & \text{whenever } p \in \mathcal{N} \\ VP_{p,p} & \longrightarrow & p & \text{whenever } p \in \mathcal{N} \\ VP_{p_2,p_1} & \longrightarrow & VP_{p_2,p_r}\ CP_{p_x,p_n}\ p_1 & \text{whenever } (p_r \leftarrow p_x; p_n \leftarrow p_1) \in IF \\ VP_{p_2,p_1} & \longrightarrow & VP_{p_2,p_n}\ p_1 & \text{whenever } p_n \leftarrow p_1 \in \mathcal{E}^{call} \\ CP_{p,p} & \longrightarrow & p & \text{whenever } p \in \mathcal{N} \\ CP_{p_2,p_1} & \longrightarrow & CP_{p_2,p_r}\ CP_{p_x,p_n}\ p_1 & \text{whenever } (p_r \leftarrow p_x; p_n \leftarrow p_1) \in IF \end{array}$$

VP_{p_2,p_1} is a shorthand for "valid execution paths from p_1 to p_2" and CP_{p_2,p_1} abbreviates "complete execution paths from p_1 to p_2". A string

generated by CP_{p_2,p_1} is an execution path from p_1 to p_2 in which every procedure-entry operation is matched by its corresponding procedure-exit operation. It is an execution path for a successful execution of the calls between p_1 and p_2. Observe that CP_{p_2,p_1} exists for p_1 and p_2 only if p_1 and p_2 are associated with the same clause and p_1 appears before p_2 when $p_1 \neq p_2$. A string generated by VP_{p_2,p_1} is an execution path from p_1 to p_2 in which every procedure-exit operation is matched by its corresponding procedure-entry operation while un-matched procedure-entry operations come from uncompleted procedure calls. If $p_1 = p_2$ then the only string generated by VP_{p_2,p_1} is p_1. It is generated by an instance of the first rule template for VP_{p_2,p_1}. If $p_1 \neq p_2$ then p_2 is reached either after or before the execution of the call $\mathbb{A}(p_1)$ (in some substitution) is completed. The string is generated by an instance of the second rule template for VP_{p_2,p_1} in the former case and by an instance of the third rule template in the later case. Finally, the rule template for VP ensures that only valid execution paths starting with the initial program point ι are generated. Let $L(X)$ denote the language generated by a non-terminal X. The set of execution paths starting with ι is $\Delta = L(VP)$ and the set of complete execution paths starting with entry points of the clauses in the program is $\Delta_c = \bigcup\{L(CP_{p,entry(C)}) \mid (C \in P) \wedge (p \in \mathcal{N}_C)\}$. All paths in the sequel are execution paths and the word "execution" will be dropped from the term "execution path".

EXAMPLE 2.3. *The grammar that generates paths for the program in example 2.1 includes among others the following rules.*

$$\begin{aligned}
VP_{6,1} &\longrightarrow VP_{6,2}\ CP_{4,4}\ 1\\
VP_{6,1} &\longrightarrow VP_{6,2}\ CP_{6,5}\ 1\\
VP_{6,1} &\longrightarrow VP_{6,5}\ 1\\
CP_{2,1} &\longrightarrow CP_{2,2}\ CP_{4,4}\ 1\\
CP_{2,1} &\longrightarrow CP_{2,2}\ CP_{6,5}\ 1
\end{aligned}$$

∎

Let $hd(\delta)$ be the leftmost point in a path δ and $tl(\delta)$ the result of removing $hd(\delta)$ from δ. Observe that if $\delta \in L(VP_{p_2,p_1})$ and $p_2 \in \mathcal{N}_C$ then there are unique $\omega \in L(CP_{p_2,entry(C)})$ and ϕ such that (a) $\delta = \omega\phi$ and (b) either $\phi = \Lambda$ or $\phi \in L(VP_{hd(\phi),p_1})$ and $entry(C) \leftarrow hd(\phi) \in \mathcal{E}^{call}$. Thus, for a path $\delta \in \Delta$, there are unique $\omega \in \Delta_c$ and $\phi \in \Delta \cup \{\Lambda\}$ such that $\delta = \omega\phi$. The prefix ω is denoted $closed\_pref(\delta)$ and the suffix ϕ is denoted $open\_suf(\delta)$. If $\phi \neq \Lambda$ then $\phi = \omega'\phi'$ for unique $\omega' \in \Delta_c$ and $\phi' \in \Delta \cup \{\Lambda\}$. In this way, a path in Δ may be decomposed into a concatenation of complete paths in Δ_c.

EXAMPLE 2.4. *Continue with examples 2.1–2.3.*
Let $\delta = 6\ 4\ 5\ 5\ 2\ 6\ 6\ 4\ 5\ 5\ 1\ 7$. *Then* δ *is the concatenation of* $\omega_1 = closed\_pref(\delta) = 6\ 4\ 5$ *and* $\delta_1 = open\_suf(\delta) = 5\ 2\ 6\ 6\ 4\ 5\ 5\ 1\ 7$. *The path* δ_1 *is decomposed into* $\omega_2 = closed\_pref(\delta_1) = 5$ *and* $\delta_2 = open\_suf(\delta_1) = 2\ 6\ 6\ 4\ 5\ 5\ 1\ 7$, *which in turn is decomposed into* $\omega_3 = closed\_pref(\delta_2) = 2\ 6\ 6\ 4\ 5\ 5\ 1$ *and* $\delta_3 = open\_suf(\delta_2) = 7$. *Finally,* $\omega_4 = open\_suf(\delta_3) = 7$ *and* $\delta_4 = closed\_pref(\delta_3) = \Lambda$. *In this way, the path* δ *is decomposed into four segments* ω_1, ω_2, ω_3 *and* ω_4. ∎

Information about paths are used in the abstract semantics to distinguish descriptions of substitutions obtained from different paths. The next section instruments LD resolution so as to make explicit the derivation path of a goal from the query. The instrumented LD resolution forms a basis for the collecting semantics in section 4. Then the abstract semantics is obtained in section 5 by approximating the collecting semantics.

3. Operational semantics

LD resolution keeps reducing one goal into another. The transition rules for one step LD derivation are

$$\frac{}{(\blacksquare\ G) \xrightarrow{LD} G}$$

and

$$\frac{\begin{array}{rl}(H \leftarrow B) \in P & \rho \in \mathsf{Ren} \\ vars((A,G)) \cap vars(\rho(H \leftarrow B)) = \emptyset & \theta = mgu(A, \rho(H)) \neq fail\end{array}}{(AG) \xrightarrow{LD} \theta((\rho(B)G))}$$

where A is an atom and G is a goal. LD resolution abstracts away some aspects of program execution that are essential to data flow analysis; it must be modified before it can be used as a basis for data flow analysis. Firstly, a substitution in which an atom in the program is executed must be made explicit because what of interest in data flow analysis is the substitution itself but not the instance of the atom under the substitution. Secondly, the derivation path of a goal from the query need be carried so as to make data flow analysis path dependent. These considerations leads to an instrumented LD resolution (ILD in abbreviation).

In order to make the presentation of ILD resolution easy to follow, we first present a modified LD resolution (MLD in abbreviation)

that incorporates the first modification. A state in MLD is a sequence $(B'_n, H_n, \theta_n) \cdots (B'_1, H_1, \theta_1)$ where B'_i is a suffix of the body of a clause whose head is H_i and θ_i is a substitution that constrains variables in the clause. The most recent clause corresponds to the front triple of the sequence and the query to the tail triple. The initial MLD state for a query $\square \leftarrow \theta_\iota(Q)$ is $(Q, \theta_\iota, \square)$. The one step MLD derivation is modelled by these two transition rules:

$$\frac{(H \leftarrow B) \in P \qquad \theta = \mathit{unify}(A, \sigma, H, \epsilon) \neq \mathit{fail}}{((A, B'), H', \sigma)s \overset{MLD}{\longrightarrow} (B, H, \theta)((A, B'), H', \sigma)s}$$

$(\blacksquare, \sigma, H)((A, B'), H', \omega)s \overset{MLD}{\longrightarrow} (B', H', \theta)s$ *where* $\theta = \mathit{unify}(H, \sigma, A, \omega)$ where $\mathit{unify} : \mathsf{Atom} \times \mathsf{Sub} \times \mathsf{Atom} \times \mathsf{Sub} \mapsto \mathsf{Sub}_{\mathit{fail}}$ is defined as

$$\mathit{unify}(A_1, \theta, A_2, \omega) \overset{def}{=} \begin{cases} \mathit{let}\ \rho \in \mathit{ren}(\theta(A_1), \omega(A_2) \\ \mathit{in} \\ \mathit{mgu}(\rho(\theta(A_1)), \omega(A_2)) \circ \omega \end{cases} \tag{1}$$

The first rule performs a procedure-entry operation. It is applied when a clause $H \leftarrow B$ in the program is resolved with a call $\sigma(A)$ where A is an atom in the program and σ a substitution. A sub-refutation for $\theta(B)$ is started where $\theta = \mathit{unify}(A, \sigma, H, \epsilon)$ is the most general unifier of H and a renaming instance of $\sigma(A)$. Note that $\sigma(A)$ instead of $H \leftarrow B$ is renamed by the operation *unify*. This is to ensure that the domain of θ contains variables in $H \leftarrow B$ instead of their renaming instances. The second rule performs a procedure-exit operation. It is applied when the first triple is of the form $(\blacksquare, \sigma, H)$ which signals the completion of a sub-refutation in which the head of the first clause is H. The second triple $((A, B'), \omega, H')$ tells that $\omega(A)$ is the call with which the clause was resolved. Thus, a sub-refutation for $\omega(A)$ has been completed. The substitution after the sub-refutation is the composition of ω and the most general unifier of $\omega(A)$ and a renaming instance of $\sigma(H)$. Observe that the call $\omega(A)$ is removed upon procedure-exit rather than procedure-entry. The operation *unify* is used in both transition rules to compute the substitution in the first triple of the next state. In other words, it encapsulates both procedure-entry and procedure exit operations.

EXAMPLE 3.1. *Consider the program in 2.1. Let "both" and "member" be abbreviated as b and m respectively. The following is a MLD refutation for query* $\square \leftarrow \theta_\iota(b(X, L1, L2))$ *with* $\theta_\iota = \{L1/[1, 2], L2/[2]\}$.

$(b(X, L1, L2), \{L1/[1, 2], L2/[2]\}, \square)$

$\overset{MLD}{\longrightarrow}_{C_1}$ $((m(X,L),m(X,K)),\{X/X_1,L/[1,2],K/[2]\},b(X,L,K))$
$(b(X,L1,L2),\{L1/[1,2],L2/[2]\},\square)$

$\overset{MLD}{\longrightarrow}_{C_3}$ $(m(X,L),\{X/X_2,H/1,L/[2]\},m(X,[H|L]))$
$((m(X,L),m(X,K)),\{X/X_1,L/[1,2],K/[2]\},b(X,L,K))$
$(b(X,L1,L2),\{L1/[1,2],L2/[2]\},\square)$

$\overset{MLD}{\longrightarrow}_{C_2}$ $(\blacksquare,\{X/2,L/[\,]\},m(X,[X|L]))$
$(m(X,L),\{X/X_2,H/1,L/[2]\},m(X,[H|L]))$
$((m(X,L),m(X,K)),\{X/X_1,L/[1,2],K/[2]\},b(X,L,K))$
$(b(X,L1,L2),\{L1/[1,2],L2/[2]\},\square)$

$\overset{MLD}{\longrightarrow}$ $(\blacksquare,\{X/2,X_2/2,H/1,L/[2]\},m(X,[H|L]))$
$((m(X,L),m(X,K)),\{X/X_1,L/[1,2],K/[2]\},b(X,L,K))$
$(b(X,L1,L2),\{L1/[1,2],L2/[2]\},\square)$

$\overset{MLD}{\longrightarrow}$ $(m(X,K),\{X/2,X_1/2,L/[1,2],K/[2]\},b(X,L,K))$
$(b(X,L1,L2),\{L1/[1,2],L2/[2]\},\square)$

$\overset{MLD}{\longrightarrow}_{C_2}$ $(\blacksquare,\{X/2,L/[\,]\},m(X,[X|L]))$
$(m(X,K),\{X/2,X_1/2,L/[1,2],K/[2]\},b(X,L,K))$
$(b(X,L1,L2),\{L1/[1,2],L2/[2]\},\square)$

$\overset{MLD}{\longrightarrow}$ $(\blacksquare,\{X/2,X_1/2,L/[1,2],K/[2]\},b(X,L,K))$
$(b(X,L1,L2),\{L1/[1,2],L2/[2]\},\square)$

$\overset{MLD}{\longrightarrow}$ $(\blacksquare,\{X/2,L1/[1,2],L2/[2]\},\square)$

Each procedure-entry operation is annotated with the label of the clause that is used. Sequences of triples are displayed vertically. ∎

The second modification is incorporated by carrying a derivation path with each triple in a MLD state. The path carried with the triple $(Q,\theta_\iota,\square)$ for the query $\square \leftarrow \theta_\iota(Q)$ is ι. The path carried in the first triple of the next state is always the path in the first triple of the current state prepended with the program point that will be visited next. The introduction of the derivation path makes redundant the suffix of the body of the clause and the head of the clause since they are uniquely determined by the leftmost program point in the path. Adding the derivation path to and removing the two redundant elements from each triple of a MLD state gives rise to a state in ILD resolution. Formally, a state in ILD is a stack that is a sequence of stack items. A stack item is of the form (δ,θ) where $\delta \in \Delta$ and $\theta \in \mathsf{Sub}$. The set of all possible stacks is then

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

ILD resolution is given by transition rules in Figure 2. Rule (R1) performs a procedure-entry operation and rule (R2) does a procedure-exit operation.

$$\frac{p \leftarrow q \in \mathcal{E}^{call} \qquad \theta = unify(\mathbb{A}(q), \sigma, \mathbb{H}(p), \epsilon) \neq fail}{(q\delta', \sigma)S \xrightarrow{ILD} (pq\delta', \theta)(q\delta', \sigma)S} \ (R1)$$

$$\frac{p \leftarrow q \in \mathcal{E}^{ret} \qquad \theta = unify(\mathbb{H}(q), \sigma, \mathbb{A}(p\text{-}), \omega) \neq fail}{(q\delta' p\text{-}\delta'', \sigma)(p\text{-}\delta'', \omega)S \xrightarrow{ILD} (pq\delta' p\text{-}\delta'', \theta)S} \ (R2)$$

Figure 2. Transition rules

EXAMPLE 3.2. *Continue with example 3.1. Below is an ILD refutation for the same query that is* $\square \leftarrow \theta_\iota(b(X, L1, L2))$ *with* $\theta_\iota = \{L1/[1,2], L2/[2]\}$. *Since the suffix and the head of the clause in a stack item are implicit, the reader need to refer to the program to follow the refutation process.*

$(7, \{L1/[1,2], L2/[2]\})$

$\xrightarrow{ILD}_{C_1}$ $(1\ 7, \{X/X_1, L/[1,2], K/[2]\})$
$(7, \{L1/[1,2], L2/[2]\})$

$\xrightarrow{ILD}_{C_3}$ $(5\ 1\ 7, \{X/X_2, H/1, L/[2]\})$
$(1\ 7, \{X/X_1, L/[1,2], K/[2]\})$
$(7, \{L1/[1,2], L2/[2]\})$

$\xrightarrow{ILD}_{C_2}$ $(4\ 5\ 1\ 7, \{X/2, L/[\ ]\})$
$(5\ 1\ 7, \{X/X_2, H/1, L/[2]\})$
$(1\ 7, \{X/X_1, L/[1,2], K/[2]\})$
$(7, \{L1/[1,2], L2/[2]\})$

$\xrightarrow{ILD}$ $(6\ 4\ 5\ 1\ 7, \{X/2, X_2/2, H/1, L/[2]\})$
$(1\ 7, \{X/X_1, L/[1,2], K/[2]\})$
$(7, \{L1/[1,2], L2/[2]\})$

$\xrightarrow{ILD}$ $(2\ 6\ 4\ 5\ 1\ 7, \{X/2, X_1/2, L/[1,2], K/[2]\})$
$(7, \{L1/[1,2], L2/[2]\})$

$\xrightarrow{ILD}_{C_2}$ $(4\ 2\ 6\ 5\ 1\ 7, \{X/2, L/[\ ]\})$
$(2\ 6\ 4\ 5\ 1\ 7, \{X/2, X_1/2, L/[1,2], K/[2]\})$
$(7, \{L1/[1,2], L2/[2]\})$

$\xrightarrow{ILD}$ $(3\ 4\ 2\ 6\ 4\ 5\ 1\ 7, \{X/2, X_1/2, L/[1,2], K/[2]\})$
$(7, \{L1/[1,2], L2/[2]\})$

$\xrightarrow{ILD}$ $(8\ 3\ 4\ 2\ 6\ 4\ 5\ 1\ 7, \{X/2, L1/[1,2], L2/[2]\})$

∎

ILD resolution is equivalent to LD resolution in the sense that, given the same goal and the same program, ILD reaches a program

point iff LD reaches the same program point, and the instantiation of the variables in the clause of the program point by ILD is equivalent (modulo renaming) to that by LD.

LEMMA 3.3. *Let θ be a substitution, C a clause and p be a program point such that $p \in \mathcal{N}_C$. Then*

- *(a) If $\theta(Q) \xrightarrow{LD}_* (\sigma(\mathbb{A}(p))G)$ for some $\sigma \in \mathsf{Sub}$ and $G \in \mathsf{Goal}$ then there are $\delta \in \Delta$, $\eta \in \mathsf{Sub}$ and $s \in \mathcal{S}$ such that $(\iota, \theta) \xrightarrow{ILD}_* (p\delta, \eta)s$ and $\sigma \uparrow vars(C) = \eta \uparrow vars(C)$.*
- *(b) If $(\iota, \theta) \xrightarrow{ILD}_* (p\delta, \eta)s$ for some $\delta \in \Delta$, $\eta \in \mathsf{Sub}$ and $s \in \mathcal{S}$ then there are $\sigma \in \mathsf{Sub}$ and $G \in \mathsf{Goal}$ such that $\theta(Q) \xrightarrow{LD}_* (\sigma(\mathbb{A}(p))G)$ and $\sigma \uparrow vars(C) = \eta \uparrow vars(C)$.*

∎

The set $\mathcal{S}_0 \subseteq \mathcal{S}$ of initial states is determined by the set of queries to the program.

$$\mathcal{S}_0 \stackrel{def}{=} \{(\iota, \theta) \mid \theta \in \Theta_\iota\}$$

The operational semantics of the program is defined as the set of descendant states of initial states where $\xrightarrow{ILD}_*$ is the reflexive and transitive closure of $\xrightarrow{ILD}$.

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

Note that the domain of the operational semantics is $\langle \wp(\mathcal{S}), \subseteq, \emptyset, \mathcal{S}, \cap, \cup \rangle$ which is a complete lattice.

4. Collecting semantics

This section presents the collecting semantics. The collecting semantics abstracts away the sequential relation between stack items of a stack. It maps a path to a set of substitutions. The domain of the collecting semantics is $\mathcal{D}^\sharp \stackrel{def}{=} \Delta \mapsto \wp(\mathsf{Sub})$ ordered by $\sqsubseteq^\sharp \in \wp(\mathcal{D}^\sharp \times \mathcal{D}^\sharp)$:

$$X^\sharp \sqsubseteq^\sharp Y^\sharp \stackrel{def}{=} \forall \delta \in \Delta.(X^\sharp(\delta) \subseteq Y^\sharp(\delta))$$

that is the pointwise extension of $\subseteq$. Since $< \wp(\mathsf{Sub}), \subseteq, \emptyset, \mathsf{Sub}, \cap, \cup >$ is a complete lattice, $< \mathcal{D}^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcap^\sharp, \sqcup^\sharp >$ is a complete lattice where $X^\sharp \sqcup^\sharp Y^\sharp = \lambda\delta \in \Delta.(X^\sharp(\delta) \cup Y^\sharp(\delta))$, $X^\sharp \sqcap^\sharp Y^\sharp = \lambda\delta \in \Delta.(X^\sharp(\delta) \cap Y^\sharp(\delta))$, $\top^\sharp = \lambda\delta \in \Delta.\mathsf{Sub}$ and $\bot^\sharp = \lambda\delta \in \Delta.\emptyset$.

The approximation of a set of stacks by an element in $\mathcal{D}^\sharp$ is modelled by a function $\gamma^\sharp \in \mathcal{D}^\sharp \mapsto \wp(\mathcal{S})$ defined as

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

where $suf(\delta)$ is the set of all suffixes of δ and $suf(\Phi) = \bigcup_{\phi \in \Phi} suf(\phi)$ for $\Phi \subseteq \Delta$.

LEMMA 4.1. *The function $\gamma^\sharp$ is a complete meet-morphism.* ∎

The collecting semantics defined below maps a path to a set of substitutions. A path is extended by performing either a procedure-entry operation or a procedure-exit operation. A path of the form $q\delta'$ is extended to $pq\delta'$ by a procedure-entry operation only if $p \leftharpoonup q \in \mathcal{E}^{call}$. A path of the form $q\zeta$ is extended to $pq\zeta$ by a procedure-exit operation only if $p \leftharpoonup q \in \mathcal{E}^{ret}$ and a prefix of $pq\zeta$ is a complete execution path for $\mathbb{A}(p\text{-})$ which is the atom to the left of the program point p. This implies that there are unique δ' and δ'' such that $pq\zeta = pq\delta' p\text{-}\delta''$ and $pq\delta' p\text{-} \in L(CP_{p,p}\text{-})$. The condition $pq\delta' p\text{-} \in L(CP_{p,p}\text{-})$ is equivalent to $q\delta' \in \Delta_c$ given that $p \leftharpoonup q \in \mathcal{E}^{ret}$. The collecting semantics is

$$[\![P]\!]^\sharp \stackrel{def}{=} lfp\ F^\sharp$$

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

$$F^\sharp(X^\sharp) \stackrel{def}{=} F_0^\sharp(X^\sharp) \sqcup^\sharp F_1^\sharp(X^\sharp) \sqcup^\sharp F_2^\sharp(X^\sharp)$$

and

$$F_0^\sharp(X^\sharp)(\delta) \stackrel{def}{=} \bigcup \{\Theta_\iota \mid (\delta = \iota)\} \quad (3)$$

$$F_1^\sharp(X^\sharp)(\delta) \stackrel{def}{=} \bigcup \left\{ cunify(\mathbb{A}(q), X^\sharp(q\delta'), \mathbb{H}(p), \{\epsilon\}) \middle| \begin{array}{c} \delta = pq\delta' \\ \wedge \\ p \leftharpoonup q \in \mathcal{E}^{call} \end{array} \right\} \quad (4)$$

$$F_2^\sharp(X^\sharp)(\delta) \stackrel{def}{=}$$
$$\bigcup \left\{ cunify(\mathbb{H}(q), X^\sharp(q\delta' p\text{-}\delta''), \mathbb{A}(p\text{-}), X^\sharp(p\text{-}\delta'')) \middle| \begin{array}{c} \delta = pq\delta' p\text{-}\delta'' \\ \wedge \\ p \leftharpoonup q \in \mathcal{E}^{ret} \\ \wedge \\ q\delta' \in \Delta_c \end{array} \right\} \quad (5)$$

and $cunify : \mathsf{Atom} \times \wp(\mathsf{Sub}) \times \mathsf{Atom} \times \wp(\mathsf{Sub}) \mapsto \wp(\mathsf{Sub})$ is

$$cunify(A_1, \Theta, A_2, \Omega) \stackrel{def}{=} \{unify(A_1, \theta, A_2, \omega) \mid \theta \in \Theta \wedge \omega \in \Omega\} \setminus \{fail\} \quad (6)$$

For a given input $X^\sharp \in \mathcal{D}^\sharp$, $F^\sharp(X^\sharp)$ maps a path to a set of substitutions. The set of substitutions for the initial path ι is $F_0^\sharp(X^\sharp)(\iota)$; that for a path δ ended with a procedure-entry operation is $F_1^\sharp(X^\sharp)(\delta)$; and that for a path δ ended with a procedure-exit operation is $F_2^\sharp(X^\sharp)(\delta)$. For a given path δ, $F_0^\sharp(X^\sharp)(\delta)$ and $F_1^\sharp(X^\sharp)(\delta)$ and $F_2^\sharp(X^\sharp)(\delta)$ are pairwise disjoint. Note that if $F_\jmath^\sharp(X^\sharp)(\delta) = \bigcup \mathcal{Y}$ with $0 \leq \jmath \leq 2$ then $\mathcal{Y}$ is a singleton because there is exactly one decomposition of δ that satisfies the predicate of $\mathcal{Y}$. The advantages of using the set union operation are two-fold. Firstly, it allows concise definitions for $F_0^\sharp$, $F_1^\sharp$ and $F_2^\sharp$. Secondly, it makes the collecting semantics correspond to the abstract semantics in form and thereby simplifies the correctness proof of the abstract semantics.

EXAMPLE 4.2. *Consider the program in example 2.1. We have* $\mathbb{H}(4) = member(X, [X|L])$ *and* $\mathbb{A}(5) = member(X, L)$ *and*

$$\begin{aligned} F^\sharp(X^\sharp)(7) &= \Theta_7 \\ &\vdots \\ F^\sharp(X^\sharp)(4\ 5\ 1\ 7) &= cunify(\mathbb{A}(5), X^\sharp(5\ 1\ 7), \mathbb{H}(4), \{\epsilon\}) \\ F^\sharp(X^\sharp)(6\ 4\ 5\ 1\ 7) &= cunify(\mathbb{H}(4), X^\sharp(4\ 5\ 1\ 7, \mathbb{A}(5), X^\sharp(5\ 1\ 7)) \\ &\vdots \end{aligned}$$

∎

The function $F^\sharp$ is monotone on $< \mathcal{D}^\sharp, \sqsubseteq^\sharp >$. The next result proves correctness of the collecting semantics.

LEMMA 4.3. $[\![P]\!] \subseteq \gamma^\sharp([\![P]\!]^\sharp)$. ∎

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. It is a mapping from paths to sets 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 $\Delta^\flat$ of path descriptions. Let $\beta : \Delta \mapsto \Delta^\flat$ maps a path into its

description. Define $\beta^{-1} : \Delta^\flat \mapsto \wp(\Delta)$ as $\beta^{-1}(\bar{\delta}) \stackrel{def}{=} \{\delta \mid \beta(\delta) = \bar{\delta}\}$ and $p \bullet \bar{\delta} \stackrel{def}{=} \{\beta(p\delta) \mid \delta \in \beta^{-1}(\bar{\delta}) \wedge p\delta \in \Delta\}$. Let $\bar{\phi} \ll \bar{\chi}$ denote the condition that at least one path described by $\bar{\chi}$ is an extension of a path described by $\bar{\phi}$ with a complete path in Δ_c, i.e., $\bar{\phi} \ll \bar{\chi} \stackrel{def}{=} \exists \phi \in \beta^{-1}(\bar{\phi}).\exists \chi \in \beta^{-1}(\bar{\chi}).\exists \delta \in \Delta_c.(\chi = \delta\phi)$. The set of the descriptions of the paths ending at p is $\Delta^\flat(p) \stackrel{def}{=} \{\beta(p\delta) \mid p\delta \in \Delta\}$.

5.2. Abstracting substitutions

When a program is analysed, the set of substitutions associated with a path is approximated by an abstract substitution. Let ASub be the domain of abstract substitutions and $\gamma \in \mathsf{ASub} \mapsto \wp(\mathsf{Sub})$ the function that gives meaning to an abstract substitution. It is required that,

C1: $< \mathsf{ASub}, \sqsubseteq, \bot, \top, \sqcap, \sqcup >$ is a complete lattice where $\sqsubseteq$ is the order on ASub, $\bot$ the infimum, $\top$ the supremum, $\sqcap$ the greatest lower bound operator and $\sqcup$ the least upper bound operator; and

C2: $\gamma \in \mathsf{ASub} \mapsto \wp(\mathsf{Sub})$ is a complete meet-morphism.

The domain of the abstract semantics is constructed in the same way as that of the collecting semantics. Let $\mathcal{D}^\flat \stackrel{def}{=} \Delta^\flat \mapsto \mathsf{ASub}$ and $\sqsubseteq^\flat \in \wp(\mathcal{D}^\flat \times \mathcal{D}^\flat)$ be defined as

$$X^\flat \sqsubseteq^\flat Y^\flat \stackrel{def}{=} \forall \bar{\delta} \in \Delta^\flat.(X^\flat(\bar{\delta}) \sqsubseteq Y^\flat(\bar{\delta}))$$

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

$$\begin{aligned} \bot^\flat &= \lambda \bar{\delta} \in \Delta^\flat.\bot \\ \top^\flat &= \lambda \bar{\delta} \in \Delta^\flat.\top \\ X^\flat \sqcap^\flat Y^\flat &= \lambda \bar{\delta} \in \Delta^\flat.(X^\flat(\bar{\delta}) \sqcap Y^\flat(\bar{\delta})) \\ X^\flat \sqcup^\flat Y^\flat &= \lambda \bar{\delta} \in \Delta^\flat.(X^\flat(\bar{\delta}) \sqcup Y^\flat(\bar{\delta})) \end{aligned}$$

The correspondence between the domains of the collecting and the abstract semantics is formalised via a function $\gamma^\flat : \mathcal{D}^\flat \mapsto \mathcal{D}^\sharp$ that is defined in terms of γ and β.

$$\gamma^\flat(X^\flat) \stackrel{def}{=} \lambda \delta \in \Delta.\gamma(X^\flat(\beta(\delta))) \tag{7}$$

LEMMA 5.1. *The function $\gamma^\flat$ is a complete meet-morphism.* ∎

5.3. Abstract Semantics

The abstract semantics is obtained as follows. A set Θ of substitutions in the collecting semantics is replaced by the best abstract substitution that approximates Θ. The concrete function *cunify* is replaced by an abstract function *aunify* that approximates safely *cunify*. A path is replaced by a path description. The data descriptions for those paths that have the same path description are merged using the least upper bound operation $\sqcup$. Let $\pi_\iota \in \mathsf{ASub}$ be the least abstract substitution such that $\Theta_\iota \subseteq \gamma(\pi_\iota)$. Note that π_ι instead of Θ_ι is given as an analysis input. Let $\mathsf{id} \in \mathsf{ASub}$, called an abstract identity substitution in [4], be the least abstract substitution such that $\epsilon \in \gamma(\mathsf{id})$. The abstract semantics is

$$[\![P]\!]^\flat \stackrel{def}{=} lfp\ F^\flat$$

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

$$F^\flat(X^\flat) \stackrel{def}{=} F_0^\flat(X^\flat) \sqcup^\flat F_1^\flat(X^\flat) \sqcup^\flat F_2^\flat(X^\flat)$$

and

$$F_0^\flat(X^\flat)(\bar{\delta}) \stackrel{def}{=} \bigsqcup\{\pi_\iota \mid \bar{\delta} = \beta(\iota)\} \tag{8}$$

$$F_1^\flat(X^\flat)(\bar{\delta}) \stackrel{def}{=} \bigsqcup \left\{ aunify(\mathbb{A}(q), X^\flat(\bar{\chi}), \mathbb{H}(p), \mathsf{id}) \;\middle|\; \begin{array}{c} q \in \mathcal{N} \\ \wedge \\ \bar{\chi} \in \Delta^\flat(q) \\ \wedge \\ p \leftharpoonup q \in \mathcal{E}^{call} \\ \wedge \\ \bar{\delta} \in (p \bullet \bar{\chi}) \end{array} \right\} \tag{9}$$

$$F_2^\flat(X^\flat)(\bar{\delta}) \stackrel{def}{=}$$

$$\bigsqcup \left\{ aunify(\mathbb{H}(q), X^\flat(\bar{\chi}), \mathbb{A}(p^-), X^\flat(\bar{\phi})) \;\middle|\; \begin{array}{c} q \in \mathcal{N} \\ \wedge \\ \bar{\chi} \in \Delta^\flat(q) \\ \wedge \\ p \leftharpoonup q \in \mathcal{E}^{ret} \\ \wedge \\ \bar{\delta} \in (p \bullet \bar{\chi}) \\ \wedge \\ \bar{\phi} \in \Delta^\flat(p^-) \\ \wedge \\ \bar{\phi} \ll \bar{\chi} \end{array} \right\} \tag{10}$$

The function $F_0^\flat(X^\flat)$ maps the path description of the initial path ι to π_ι and other path descriptions to $\bot$. The function $F_1^\flat(X^\flat)$ accumulates, for each path description, data descriptions from procedure-entry

operations. If the path description does not describes any path whose last operation is a procedure-entry, $F_1^\flat(X^\flat)$ maps that path description to $\bot$. Similarly, the function $F_2^\flat(X^\flat)$ accumulates, for each path description, data descriptions from procedure-exit operations. If the path description does not describes any path whose last operation is a procedure-exit, $F_2^\flat(X^\flat)$ maps that path description to $\bot$. The semantic function $F^\flat$ is monotone on $< \mathcal{D}^\flat, \sqsubseteq^\flat >$ if *aunify* is monotone in its second and fourth arguments. The global correctness of the abstract semantics is ensured by the following two local correctness conditions.

C3: $\epsilon \in \gamma(\mathsf{id})$.

C4: $cunify(A_1, \gamma(\pi_1), A_2, \gamma(\pi_2)) \subseteq \gamma \circ aunify(A_1, \pi_1, A_2, \pi_2)$ for any $\pi_1 \in \mathsf{ASub}$, any $\pi_2 \in \mathsf{ASub}$, any $A_1 \in \mathsf{Atom}$ and any $A_2 \in \mathsf{Atom}$.

The following theorem proves that C1-C4 are sufficient conditions for the abstract semantics to approximate safely the collecting semantics.

THEOREM 5.2. If C1-C4 hold then $[\![P]\!]^\sharp \sqsubseteq^\sharp \gamma^\flat([\![P]\!]^\flat)$. ∎

Note that the conditions C1-C4 are exactly those required by the abstract semantics in [43, 31]. 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

This section shows that the abstract semantics can be instantiated for different abstractions of paths. The simplest 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 $p \leftharpoonup q \in \mathcal{E}$ and $p^- \ll q$ if $p \leftharpoonup q \in \mathcal{E}^{ret}$. In this case, the abstract semantics degenerates to that in [43].

6.1. Call Strings

Call strings have been used to enhance analysis of programs of other programming paradigms [50]. The idea is to keep track of calls on the execution stack - calls that are currently being executed. This amounts to contracting each complete path segment in the decomposition of a

path to the end point of the segment. The call string of a path is given by a function $call : \Delta \mapsto \mathcal{N}^*$ defined as

$$call(\delta) \stackrel{def}{=} \begin{cases} hd(closed\_pref(\delta)) & \text{If } open\_suf(\delta) = \Lambda; \\ hd(closed\_pref(\delta))call(open\_suf(\delta)) & \text{Otherwise.} \end{cases}$$

Since there might be stacks of infinite size due to recursion, it is usual to keep track of top k calls with $k \geq 1$. This can be achieved by defining $\beta(\delta) \stackrel{def}{=} [call(\delta)]_k$ where $[\delta']_{k'}$ is the result of truncating δ' at position $k'+1$ for $k' \geq 1$ and $[\delta']_0 = \Lambda$. A call string will be written in the form of $p\bar{\delta}$ since a call string has at least one program point. Let $\Delta^\flat \stackrel{def}{=} \{\beta(\delta) \mid \delta \in \Delta\}$. If $p \leftharpoonup q \in \mathcal{E}^{call}$ and $q\bar{\chi} \in \Delta^\flat(q)$ then $p \bullet (q\bar{\chi}) = \{p[q\bar{\chi}]_{k-1}\}$. If $p \leftharpoonup q \in \mathcal{E}^{ret}$ and $q\bar{\chi} \in \Delta^\flat(q)$ then (a) $p\bar{\delta} \in (p \bullet (q\bar{\chi}))$ iff $\bar{\chi} = [p\text{-}\bar{\delta}]_{k-1}$ and (b) $p\text{-}\bar{\phi} \in \Delta^\flat(p\text{-})$ and $p\text{-}\bar{\phi} \ll q\bar{\chi}$ imply $\bar{\phi} = \bar{\delta}$. Thus, $F^\flat$ is specialised into the following.

$$F^\flat(X^\flat)(p\bar{\delta}) \stackrel{def}{=}$$
$$\begin{cases} \pi_\iota, & \text{if } (p = \iota) \\ \bigsqcup \left\{ aunify(\mathbb{A}(q), X^\flat(q\bar{\chi}), \mathbb{H}(p), \mathsf{id}) \middle| \begin{array}{c} p \leftharpoonup q \in \mathcal{E}^{call} \\ \wedge \\ \bar{\delta} = [q\bar{\chi}]_{k-1} \end{array} \right\}, & \text{if } p \in \mathcal{N}^{call} \\ \bigsqcup \left\{ aunify(\mathbb{H}(q), X^\flat(q\bar{\chi}), \mathbb{A}(p\text{-}), X^\flat(p\text{-}\bar{\delta})) \middle| \begin{array}{c} p \leftharpoonup q \in \mathcal{E}^{ret} \\ \wedge \\ \bar{\chi} = [p\text{-}\bar{\delta}]_{k-1} \end{array} \right\}, & \text{if } p \in \mathcal{N}^{ret} \end{cases}$$

Note that if $p\bar{\delta} \in \Delta^\flat$ and $p = \iota$ then $\bar{\delta} = \Lambda$. The abstract semantics in [43] is a special case of the above abstract semantics for $k = 1$.

EXAMPLE 6.1. *Let $k = 3$. The program in example 2.1 has 19 call strings of length 3 or less. Among them are* 5 5 1, 5 5 2, 5 5 5, 4 5 5 *and* 6 5 5. *The abstract semantics $F^\flat(X^\flat)$ is defined by 19 equations one for each of the 19 call strings. The following are three of these equations.*

$$\begin{aligned} F^\flat(X^\flat)(7) &= \pi_7 \\ F^\flat(X^\flat)(5\ 5\ 5) &= aunify(\mathbb{A}(5), X^\flat(5\ 5\ 1), \mathbb{H}(5), \mathsf{id}) \\ &\quad \sqcup \\ &\quad aunify(\mathbb{A}(5), X^\flat(5\ 5\ 2), \mathbb{H}(5), \mathsf{id}) \\ F^\flat(X^\flat)(6\ 5\ 5) &= aunify(\mathbb{H}(4), X^\flat(4\ 5\ 5), \mathbb{A}(5), X^\flat(5\ 5\ 5)) \\ &\quad \sqcup \\ &\quad aunify(\mathbb{H}(6), X^\flat(6\ 5\ 5), \mathbb{A}(5), X^\flat(5\ 5\ 5)) \end{aligned}$$

∎

EXAMPLE 6.2. *Consider the program in 2.1 and call strings of length 2. Below is the result of mode analysis [4, 15] 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 [51] between variables to ensure correctness of analysis although no two variables in the same clause share in this example. The input abstract substitution π_ι, contained in the first comment, indicates that both(X,L1,L2) is called with both L1 and L2 being ground and X being free. All other abstract substitutions are inferred by the analyser.*

```
$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/ground,Y/ground],[]
    % (both/3,1),2-[L/ground,X/ground,Y/ground],[]
    % (member/2,2),1-[L/ground,X/ground,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 call string of length 1 and an abstract substitution. The program point and the call string of length 1 in a comment annotated with that program point form a call string of length 2. The empty call string is displayed as toplevel *indicating that the entry point*

of the query is reached by the language system. 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.

*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 specialise member/2 into two different versions. Without keeping track 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 call and which call invokes a given clause. This corresponds to describing a path by its last operation. Thus, $\beta(pq\delta) = pq$ and $\beta(\iota) = \iota\Lambda$. Note that $pq \in \Delta^\flat(p)$ implies $p \leftarrow q \in \mathcal{E}$ or $p = \iota \wedge q = \Lambda$. Thus, $\Delta^\flat = \{pq \mid p \leftarrow q \in \mathcal{E}\} \cup \{\iota\Lambda\}$. Observe that $p \bullet \bar{\chi} = \{pq\}$ for any $\bar{\chi} \in \Delta^\flat(q)$ and p such that $p \leftarrow q \in \mathcal{E}$ and that $\bar{\phi} \ll \bar{\chi}$ if $\bar{\phi} \in \Delta^\flat(p\text{-})$, $\bar{\chi} \in \Delta^\flat(q)$ and $p \leftarrow q \in \mathcal{E}^{ret}$. Therefore, $F^\flat$ is specialised into the following.

$$F^\flat(X^\flat)(pq) \stackrel{def}{=}$$
$$\begin{cases} \pi_\iota, & \text{if } (p = \iota) \\ \bigsqcup\{aunify(\mathbb{A}(q), X^\flat(qu), \mathbb{H}(p), \mathsf{id}) \mid q \leftarrow u \in \mathcal{E}\}, & \text{if } p \in \mathcal{N}^{call} \\ \bigsqcup \left\{ aunify(\mathbb{H}(q), X^\flat(qu), \mathbb{A}(p\text{-}), X^\flat(p\text{-}v)) \middle| \begin{array}{c} q \leftarrow u \in \mathcal{E} \\ \wedge \\ p\text{-} \leftarrow v \in \mathcal{E} \end{array} \right\}, & \text{if } p \in \mathcal{N}^{ret} \end{cases}$$

Note that $p = \iota$ and $pq \in \Delta^\flat$ imply $q = \Lambda$. If Λ is added to $\mathcal{N}$ and an edge from Λ to ι is added to $\mathcal{E}$ then the above abstract semantics associates an abstract substitution with each edge in the program graph.

EXAMPLE 6.3. *The program in example 2.1 has 15 pairs including* $7\ \Lambda$. *The abstract semantics* $F^\flat(X^\flat)$ *is defined by 15 equations one for each pair. The following are three of these equations.*

$$\begin{aligned}
F^\flat(X^\flat)(7\ \Lambda) &= \pi_7 \\
F^\flat(X^\flat)(5\ 5) &= \mathit{aunify}(\mathbb{A}(5), X^\flat(5\ 1), \mathbb{H}(5), \mathsf{id}) \\
&\quad \sqcup \\
&\quad \mathit{aunify}(\mathbb{A}(5), X^\flat(5\ 2), \mathbb{H}(5), \mathsf{id}) \\
&\quad \sqcup \\
&\quad \mathit{aunify}(\mathbb{A}(5), X^\flat(5\ 5), \mathbb{H}(5), \mathsf{id}) \\
F^\flat(X^\flat)(8\ 3) &= \mathit{aunify}(\mathbb{H}(3), X^\flat(3\ 4), \mathbb{A}(7), X^\flat(7\ \Lambda)) \\
&\quad \sqcup \\
&\quad \mathit{aunify}(\mathbb{H}(3), X^\flat(3\ 6), \mathbb{A}(7), X^\flat(7\ \Lambda))
\end{aligned}$$

∎

EXAMPLE 6.4. *This example applies the above abstract semantics to perform prescriptive type analysis [24, 1, 25, 8, 32, 28]. In a prescriptive type analysis, type definitions are given as an analysis input. The following type definitions are used.*

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

Below is a buggy naive reverse program and the result of the prescriptive type analysis of the program using the abstract domain in [24]. The program is annotated as follows. Each program point is annotated with a few comments; one for each edge to that program point. The first part of the comment for an edge is the source program point and the second part an abstract substitution. 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* π *denotes the set of those substitutions that instantiate each variable* X *in the domain of* π *into a term of the type* $\pi(X)$. bot *is the type denoting the empty set of terms and* top *is the type denoting the set of all terms. The input abstract substitution* π_ι, *contained in the first comment, indicates that nrev(X,Y) is called with* X *being a list of natural numbers. All other abstract substitutions are inferred by the analyser.*

```
$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 [43] 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, 25, 8, 32, 28, 33], [33] is the most precise one. Using a disjunction of variable typings as an abstract substitution, [33] together with the abstract semantics in [43] 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. ∎

6.3. Other path abstractions

A call string gives information about uncompleted calls and discards information about completed calls. The last transfer of control informs which call invoked a clause or which clause is used to satisfy a call. Let δ be a path and $\textcircled{p}A_{\textcircled{q}}$ be an atom in the program such that $hd(\delta) = q$, i.e. q is reached by following the path δ. The path δ has the form $q\eta p\omega$

where ω is the path before the atom A is called and η is the path segment for a successful execution of the body of the clause that was used to satisfy the atom A. Path abstractions in sections 6.1 and 6.2 can be combined to improve precision of each other. A path description will be of the form $\langle c, e\rangle$ where c is a call string of some fixed length and e is the edge for the last transfer of control. If $\langle c, e\rangle$ is the description of the path δ then c describes ω and e – a return edge – describes η. Note that a combined description for a path from the initial program point to the entry point of a clause gives the same information as the call string in the combined description.

More refined path abstractions are obtained by replacing the call string c and/or the return edge e with more detailed information about ω and η respectively. The path segment η may be described by the set of predicates that have been called during the traversal of η. Another way of describing η is to visualise η as forming a proof tree for $\textcircled{p}A\textcircled{q}$ and then abstract the proof tree using techniques surveyed in [12]. These refined abstractions of η will allow a diagnoser to trace bugs much quickly than suggested in section 6.2. Information about ω may include information about completed calls before the atom is invoked. These completed calls may be described in the same way as η is described. These refined abstractions about ω can be used to generate different versions of clauses.

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 [41, 29]. Since only Horn clause logic programs are considered in our work, there is no need for a control flow analysis.

Context information is also present in data flow analysis of logic programs although context sensitivity of logic program analysis has not been studied on its own. The abstract semantics proposed in this paper is now compared with other abstract semantics for logic programs. An abstract semantics for logic programs is based on either a top-down evaluation strategy or a bottom-up evaluation strategy or a system of simultaneous recurrence equations generated from the program.

7.1. Recurrence-based Abstract Interpretation

The abstract semantics in [39, 43] do not keep track of any context information at all. As shown in section 6, [43] is a special form of our abstract semantics. The abstract semantics in [37] 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 [37] and therefore can infer more precise results. It also separates the abstraction of paths from that of data.

The abstract semantics in [57] 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 [57] 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 complete path segments is ignored that is useful as shown in example 6.4.

7.2. Bottom-up abstract interpretation

The bottom-up abstract semantics in [2, 23, 7, 35, 36] approximate the success set of the program [54] 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, these bottom-up abstract semantics cannot make use of path information.

The abstract semantics in [26] derives demands on queries that guarantee that the execution of the program satisfies demands associated with the program. Demands can be both call patterns and success patterns. The abstract semantics is formulated as a greatest fixed point computation with a preceding least fixed point computation. The least fixed point computation approximates the success set of the program which is then used by the greatest fixed point computation to propagate demands backwards over the flow of control. The abstract semantics

in [17] derives the weakest specification for a module in a program that guarantees that the program satisfies its specification. A specification is a collection of pairs consisting of a goal and a success pattern. The abstract semantics is based on the unfolding semantics in [3] that is a bottom-up computation. The abstract semantics in [38] uses a bottom-up computation to produce a collection of optimisation opportunities for each predicate. An optimisation opportunity is a pair consisting of a call pattern and a set of program points where optimisations may be performed. The satisfaction of the call pattern indicates that optimisations at the program points are possible; and such possibilities may be verified by a further analysis. No context information is used in [26] or [17] or [38].

7.3. Top-down abstract interpretation

The abstract semantics in [4, 40, 56, 18] mimic LD resolution. [40, 56, 18] differ from [4] only in their dealing with recursive calls. The abstract semantics in [4] constructs an abstract AND-OR graph that describes all the intermediate proof trees for the queries satisfying a query description. An AND-node is (labelled with) a clause head and its child OR-nodes are (labelled 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 π in a partially constructed abstract AND-OR graph. The abstract semantics computes the abstract success substitution of A as follows. For each clause $H \leftarrow A_1 A_2 \cdots A_m \blacksquare$ such that H may match with $\theta(A)$ for some θ satisfying π, it adds to A a child AND-node H that has m child OR-nodes $A_1, \cdots, A_m$ and performs an abstract procedure-entry operation to obtain the abstract call substitution π_{in} of A_1. The abstract semantics extends A_1 recursively and extends A_{j+1} using the abstract success substitution of A_j as its abstract call substitution. After the abstract success substitution π_{out} of the last OR-node A_m has been computed for each matching clause, the abstract success substitution π_{succ} of A is obtained by performing an abstract procedure-exit operation for each of these clauses and computing an upper bound of the results.

Suppose that an OR-node A with abstract call substitution π were to be extended. If A has an ancestor OR-node A' with abstract call substitution π' such that A is a variant of A and π is a variant of π', the abstract semantics initialises 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' stabilises. 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 complete execution paths for A.

The abstract semantics in [25, 6] mimic the OLDT resolution [52]. 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

An abstract semantics is presented that is parameterised 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.

Acknowledgements

Comments and suggestions from anonymous referees on an earlier version of this paper are greatly appreciated.

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. A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A compositional semantics for logic programs. *Theor. Comput. Sci.*, 122(1-2):3–47, 1994.
4. M. Bruynooghe. A practical framework for the abstract interpretation of logic progams. *Journal of Logic Programming*, 10(2):91–124, 1991.
5. M. Bruynooghe, M. Codish, S. Genaim, and W. Vanhoof. Reuse of results in termination analysis of typed logic programs. In M. Hermenegildo and G. Puebla, editors, *Proceedings of The Ninth International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 477–492, 2002.
6. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Transaction on Programming Languages and Systems*, 16(1):35–101, 1994.
7. M. Codish, D. Dams, and E. Yardani. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science*, 124:93–125, 1994.
8. M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science*, 238:131–159, 2000.
9. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252. The ACM Press, 1977.
11. 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.
12. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Conference Record of FPCA'95 Conference on Functional Programming and Computer Architecture*, pages 170–181. The ACM Press, 1995.
13. S.K. Debray. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, 1989.
14. S.K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Transaction on Programming Languages and Systems*, 11(3):418–450, 1989.
15. S.K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–230, 1988.
16. P. Deransart, B. Lorho, and J. Małuszynski, editors. *Proceedings of the First International Workshop on Programming Language Implementation and Logic Programming*, volume 348 of *Lecture Notes in Computer Science*. Springer, 1988.
17. R. Giacobazzi. Abductive analysis of modular logic programs. *Journal of Logic and Computation*, 8(4):457–484, 1998.
18. 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.

19. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.
20. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. *ACM SIGSOFT Software Engineering Notes*, 20(4):104–115, 1995.
21. D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *Journal of Logic Programming*, 13(1–4):291–314, 1992.
22. 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.
23. T. Kanamori. Abstract interpretation based on Alexander Templates. *Journal of Logic Programming*, 15(1 & 2):31–54, 1993.
24. 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.
25. T. Kanamori and T. Kawamura. Abstract interpretation based on OLDT resolution. *Journal of Logic Programming*, 15(1 & 2):1–30, 1993.
26. A. King and L. Lu. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming*, 2(4&5):517–547, 2002. http://xxx.lanl.gov/abs/cs.PL/0201011.
27. R. A. Kowalski and K. A. Bowen, editors. *Proceedings of the Fifth International Conference and Symposium on Logic Programming*. The MIT Press, 1988.
28. 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*, volume 1490 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 1998.
29. 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.
30. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
31. L. Lu. Abstract interpretation, bug detection and bug diagnosis in normal logic programs. PhD thesis, University of Birmingham, 1994.
32. L. Lu. A polymorphic type analysis in logic programs by abstract interpretation. *Journal of Logic Programming*, 36(1):1–54, 1998.
33. 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 Practice of Declarative Programming*, pages 214–225. The ACM Press, 2000.
34. L. Lu and A. King. Type inference generalises type checking. In M. Hermenegildo and G. Puebla, editors, *Proceedings of Ninth International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 85–101, 2002.
35. K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In Kowalski and Bowen [27], pages 733–748.
36. K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of normal logic programs. *Journal of Logic Programming*, 13(1–4):181–204, 1992.
37. 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.

38. N. Mazur, G. Janssens, and W. Vanhoof. Collecting potential optimizations. In *Proceedings of International Symposium on Logic-based Program Synthesis and Transformation*, pages 115–120, 2002.
39. 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.
40. K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(1–4):315–347, 1992.
41. 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.
42. F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
43. U. Nilsson. Towards a framework for abstract interpretation of logic programs. In Deransart et al. [16], pages 68–82.
44. T. Reps. Shape analysis as a generalized path problem. In *Proceedings of the ACM SIGPLAN Symposium on Partial evaluation and semantics-based program manipulation*, pages 1–11. The ACM Press, 1995.
45. T. Reps. Program analysis via graph-reachability. In *Proceedings of the 1997 international symposium on Logic programming*, pages 5–19. The MIT Press, 1997.
46. T. Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162–186, 2000.
47. T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes*, 19(5):11–20, 1994.
48. B. G. Ryder, W. A. Landi, P. H. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, 2001.
49. D. De Schreye and M. Bruynooghe. An application of abstract interpretation in source level program transformation. In Deransart et al. [16], pages 35–57.
50. 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.
51. 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*, volume 213 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 1986.
52. 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.
53. 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.
54. M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Artificial Intelligence*, 23(10):733–742, 1976.
55. K. Verschaetse and D. De Schreye. Deriving termination proofs for logic programs, using abstract procedures. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 301–315. The MIT Press, 1991.

56. A. Waern. An implementation technique for the abstract interpretation of Prolog. In Kowalski and Bowen [27], pages 700–710.
57. 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, i.e. $rang(\theta) \stackrel{def}{=} \bigcup\{vars(\theta(x)) \mid x \in dom(\theta)\}$. Let $o_1 \cong o_2$ denote the relation $o_1 = \rho(o_2)$ for some renaming substitution ρ. Then $\cong$ is an equivalence relation. The parentheses in the application of a substitution to a term and the function composition operator $\circ$ in the composition of two substitutions will be omitted when no ambiguity arises. We also assume that $\circ$ binds stronger than $\uparrow$.

LEMMA A.1. *Let ρ be a renaming substitution 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 $\rho(\phi)$. Assume that $\phi'(a')$ and $\psi(b)$ unify. There is a substitution θ such that $\theta(\phi'(a')) = \theta(\psi(b))$. By hypothesis, $vars(a') \cap dom(\psi) = \emptyset$ 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 renaming substitutions such that*

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

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

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

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

Then

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

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

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

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

By Eq. 15-16,

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

$\theta_1\rho_3 a = \theta_1 a = \theta_1\rho_1 b = \theta_1(\rho_3\rho_2 \uparrow \mathcal{V})b = \theta_1\rho_3\rho_2 b$ 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

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

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

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

By Eq. 18-19,

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

$\theta_2\rho_4 a = \theta_2 a = \theta_2\rho_2 b = \theta_2(\rho_4\rho_1 \uparrow \mathcal{V})b = \theta_2\rho_4\rho_1 b$ 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.

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

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

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

$$\left(\begin{array}{l}\{X_{u_1}/\overline{x}_{u_1}, \cdots, X_{u_p}/\overline{x}_{u_p}\} \\ \cup \{Z_{v_o}/z_{v_o} \mid 1 \leq o \leq q \wedge Z_{v_o} \notin \mathcal{Y}\} \\ \cup \{Y_1/z_1, \cdots, Y_l/z_l\}\end{array}\right) = \tag{23}$$
$$\zeta_1(\{X_{i_1}/x_{i_1}, \cdots, X_{i_s}/x_{i_s}\} \cup \{Y_{j_1}/y_{j_1}, \cdots, Y_{j_t}/y_{j_t}\})$$
$$\left(\begin{array}{l}\{X_{i_1}/x_{i_1}, \cdots, X_{i_s}/x_{i_s}\} \\ \cup \{Y_{j_o}/y_{j_o} \mid 1 \leq o \leq t \wedge Y_{j_o} \notin \mathcal{Z}\} \\ \cup \{Z_1/y_1, \cdots, Z_l/y_l\}\end{array}\right) = \tag{24}$$
$$\zeta_2(\{X_{u_1}/\overline{x}_{u_1}, \cdots, X_{u_p}/\overline{x}_{u_p}\} \cup \{Z_{v_1}/z_{v_1}, \cdots, Z_{v_q}/z_{v_q}\})$$

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, $\{X_{i_1}, \cdots, X_{i_s}\} = \{X_{u_1}, \cdots, X_{u_p}\}$. Hence, $s = p$ and $i_o = u_o$ for $1 \leq o \leq s$. Also, $x_{i_o} = \zeta_2(\overline{x}_{i_o})$ 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,

$$Y_h/z_h \in \zeta_1 \;\; If \;\; h \notin \{j_1, \cdots, j_t\}$$
$$z_h = \zeta_1(y_h) \;\; If \;\; h \in \{j_1, \cdots, j_t\}$$

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 $1 \leq h \leq l$,

$$z_h = \zeta_1(y_h) \tag{25}$$

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

$$y_h = \zeta_2(z_h) \tag{26}$$

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 substitution 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.2.(a) and (c) by letting $\rho_2 = \rho$ and ρ_1 be a renaming substitution such that $\rho_1 X = X$ for each $X \in vars(b)$. ∎

COROLLARY A.4. *Let a_1 and a_2 be two atoms, ρ_1 and ρ_2 be renaming substitutions. If*

$$\begin{aligned} dom(\rho_1) &\supseteq vars(a_1) \\ dom(\rho_2) &\supseteq vars(a_2) \\ vars(\rho_1 a_1) \cap vars(a_2) &= \emptyset \\ vars(\rho_2 a_2) \cap vars(a_1) &= \emptyset \end{aligned}$$

then $\rho_1 a_1$ and a_2 unify iff a_1 and $\rho_2 a_2$ unify, and

$$(mgu(\rho_1 a_1, a_2)\ \rho_1) \uparrow dom(\rho_1) \cong mgu(a_1, \rho_2 a_2) \uparrow vars(a_1)$$

PROOF: We only prove the *if* part since the *only if* part is dual. Let ρ_2' be a renaming substitution such that $dom(\rho_2') = dom(\rho_2)$, $vars(\rho_2' a_2) \cap vars(a_1) = \emptyset$ and $vars(\rho_1 a_1) \cap vars(\rho_2' a_2) = \emptyset$.

Suppose a_1 and $\rho_2 a_2$ unify. By lemma A.2.(a), a_1 and $\rho_2' a_2$ unify, and

$$mgu(a_1, \rho_2' a_2) \uparrow vars(a_1) \cong mgu(a_1, \rho_2 a_2) \uparrow vars(a_1) \qquad (27)$$

by lemma A.2.(b). By corollary A.3, $\rho_1 a_1$ and $\rho_2' a_2$ unify, and

$$mgu(\rho_1 a_1, \rho_2' a_2)\ \rho_1 \uparrow vars(a_1) \cong mgu(a_1, \rho_2' a_2) \uparrow vars(a_1) \qquad (28)$$

So, by Eq. 27-28,

$$mgu(\rho_1 a_1, \rho_2' a_2)\ \rho_1 \uparrow vars(a_1) \cong mgu(a_1, \rho_2 a_2) \uparrow vars(a_1) \qquad (29)$$

By corollary A.3, $\rho_1 a_1$ and a_2 unify,

$$mgu(\rho_1 a_1, a_2) \uparrow vars(\rho_1 a_1) \cong mgu(\rho_1 a_1, \rho_2' a_2) \uparrow vars(\rho_1 a_1)$$

that implies

$$mgu(\rho_1 a_1, a_2)\ \rho_1 \uparrow vars(a_1) \cong mgu(\rho_1 a_1, \rho_2' a_2)\rho_1 \uparrow vars(a_1) \qquad (30)$$

So, $mgu(\rho_1 a_1, a_2)\rho_1 \uparrow vars(a_1) \cong mgu(a_1, \rho_2 a_2) \uparrow vars(a_1)$ by Eq. 29-30. It now suffices to prove

$$mgu(\rho_1 a_1, a_2)\ \rho_1 \uparrow dom(\rho_1) \cong mgu(\rho_1 a_1, a_2)\ \rho_1 \uparrow vars(a_1)$$

Let $\rho_1^1 = \rho_1 \uparrow vars(a_1)$ and $\rho_1^2 = \rho_1 \uparrow (dom(\rho_1) \setminus vars(a_1))$. Then $\rho_1 = \rho_1^1 \cup \rho_1^2$,

$$\begin{aligned} mgu(\rho_1 a_1, a_2)\ \rho_1 \uparrow dom(\rho_1) &= mgu((\rho_1^1 \cup \rho_1^2)a_1, a_2)\ (\rho_1^1 \cup \rho_1^2) \uparrow dom(\rho_1) \\ &= mgu(\rho_1^1 a_1, a_2)\rho_1^1 \uparrow vars(a_1) \cup \rho_1^2 \end{aligned}$$

and

$$\begin{aligned} mgu(\rho_1 a_1, a_2)\rho_1 \uparrow vars(a_1) &= mgu(a_1(\rho_1^1 \cup \rho_1^2), a_2)(\rho_1^1 \cup \rho_1^2) \uparrow vars(a_1) \\ &= mgu(\rho_1^1 a_1, a_2)\rho_1^1 \uparrow vars(a_1) \end{aligned}$$

Also, $rang(mgu(\rho_1^1 a_1, a_2)\rho_1^1 \uparrow vars(a_1)) \cap dom(\rho_1^2) = \emptyset$ and $dom(\rho_1^2) \cap vars(a_1) = \emptyset$. So,

$$\begin{aligned} (mgu(\rho_1 a_1, a_2)\rho_1 \uparrow vars(a_1))\rho_1^2 &= (mgu(\rho_1^1 a_1, a_2)\rho_1^1 \uparrow vars(a_1))\rho_1^2 \\ &= mgu(\rho_1^1 a_1, a_2)\rho_1^1 \uparrow vars(a_1) \cup \rho_1^2 \\ &= mgu(\rho_1 a_1, a_2)\rho_1 \uparrow dom(\rho_1) \end{aligned}$$

Therefore, $mgu(\rho_1 a_1, a_2)\rho_1 \uparrow dom(\rho_1) \cong mgu(\rho_1 a_1, a_2)\rho_1 \uparrow vars(a_1)$ since ρ_1^2 is a renaming substitution. ∎

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 3.3

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

Consider procedure-entry operation first. Let $\tau_q(\rho_C(\mathbb{A}(q))G)$ be a goal in LD where $\mathbb{A}(q)$ 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 ILD 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 $\tau_q(\rho_C(\mathbb{A}(q))G) \xrightarrow{LD} \tau_p(\rho_{C'}(B)\tau_q(G))$ iff $(q\delta', \sigma_q)s \xrightarrow{ILD} (pq\delta', \sigma_p)(q\delta', \sigma_q)s$ and $\sigma_p \uparrow \mathcal{V}_{C'} \cong \tau_p\rho_{C'} \uparrow \mathcal{V}_{C'}$ where $\rho_{C'}$ is the renaming substitution applied to C' in LD.

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

$$\zeta(\sigma_q \uparrow \mathcal{V}_C) = \tau_q\rho_C \uparrow \mathcal{V}_C \tag{31}$$

By corollary A.4, $\tau_q(\rho_C(\mathbb{A}(q))G) \xrightarrow{LD} \tau_p(\rho_{C'}(B)\tau_q(G))$ iff $(q\delta', \sigma_q)s \xrightarrow{ILD} (pq\delta', \sigma_p)(q\delta', \sigma_q)s$. Suppose $\tau_q(\rho_C(\mathbb{A}(q))G) \xrightarrow{LD} \tau_p(\rho_{C'}(B)\tau_q(G))$. Then

$$\begin{aligned}
&\tau_q\rho_C\mathbb{A}(q)\\
&\quad = (\tau_q\rho_C \uparrow \mathcal{V}_C)\mathbb{A}(q) \quad (\because vars(\mathbb{A}(q)) \subseteq \mathcal{V}_C)\\
&\quad = (\zeta(\sigma_q \uparrow \mathcal{V}_C))\mathbb{A}(q) \qquad (\because Eq.\ 31)\\
&\quad = \zeta\sigma_q\mathbb{A}(q) \qquad (\because vars(\mathbb{A}(q)) \subseteq \mathcal{V}_C)
\end{aligned} \tag{32}$$

and

$$\begin{aligned}
&\tau_p\rho_{C'} \uparrow \mathcal{V}_{C'}\\
&\quad = mgu(\rho_{C'}H, \tau_q\rho_C\mathbb{A}(q))\ \rho_{C'} \uparrow \mathcal{V}_{C'}\\
&\quad = mgu(\rho_{C'}H, \zeta\sigma_q\mathbb{A}(q))\ \rho_{C'} \uparrow \mathcal{V}_{C'} \quad (\because Eq.\ 32)
\end{aligned} \tag{33}$$

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

$$\begin{aligned}
&\sigma_p \uparrow \mathcal{V}_{C'}\\
&\quad = mgu(H, \psi\sigma_q\mathbb{A}(q)) \uparrow \mathcal{V}_{C'}\\
&\quad = mgu(H, \psi\overline{\zeta}\zeta\sigma_q\mathbb{A}(q)) \uparrow \mathcal{V}_{C'} \qquad (\because \overline{\zeta}\zeta\ is\ identity)\\
&\quad = mgu(H, (\psi\overline{\zeta})(\zeta\sigma_q\mathbb{A}(q))) \uparrow \mathcal{V}_{C'}\\
&\quad = mgu(H, (\psi\overline{\zeta})(\zeta\sigma_q\mathbb{A}(q))) \uparrow vars(H)
\end{aligned} \tag{34}$$

$\sigma_p \uparrow \mathcal{V}_{C'} \cong \tau_p\rho_{C'} \uparrow \mathcal{V}_{C'}$ by corollary A.4 and Eq. 33-34. This completes the first part of the proof.

Now consider procedure-exit operation. Let $r = exit(C')$, the current ILD state be $(r\delta''pq\delta', \sigma_r)(q\delta', \sigma_q)s$ and the current goal in LD be $\tau_r(G)$. Let $(r\delta''pq\delta', \sigma_r)(q\delta', \sigma_q)s \xrightarrow{ILD} (q^+r\delta''pq\delta', \sigma_{q^+})s$. 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 substitution 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'. Then $\sigma_r \uparrow \mathcal{V}_{C'} = \zeta'\tau_r\rho_{C'} \uparrow \mathcal{V}_{C'}$. Let ϕ' be a

renaming substitution and θ be the computed answer to $\tau_p(\rho_{C'}(B))$. Then, $\tau_r = \theta\tau_p$ and

$$\begin{array}{l}
\phi'\sigma_r H \\
\quad = \phi'\zeta'\tau_r\rho_{C'}H \\
\quad = \phi'\zeta'\theta\eta\tau_q\rho_{C'}H \\
\quad = \phi'\zeta'\theta\eta\rho_{C'}H \qquad (\because\ vars(\rho_{C'}C') \cap vars(\rho_C C) = \emptyset) \\
\quad = \phi'\zeta'\theta\eta\tau_q\rho_C\mathbb{A}(q) \qquad (35)
\end{array}$$

where $\eta = mgu(\rho_{C'}H, \tau_q\rho_C\mathbb{A}(q))$. By Eq. 31,

$$\sigma_q \uparrow \mathcal{V}_C = \overline{\zeta}\tau_q\rho_C \uparrow \mathcal{V}_C \qquad (36)$$

So,

$$\begin{array}{ll}
\sigma_q\mathbb{A}(q) & \\
\quad = (\overline{\zeta}\tau_q\rho_C \uparrow \mathcal{V}_C)\mathbb{A}(q) & (\because\ Eq.\ 36) \\
\quad = \overline{\zeta}\tau_q\rho_C\mathbb{A}(q) & (\because\ vars(\mathbb{A}(q)) \subseteq \mathcal{V}_C) \qquad (37)
\end{array}$$

Therefore, letting $A = \tau_q\rho_C\mathbb{A}(q)$,

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

This completes the proof of the lemma.

A.3. Proof of lemma 4.1

Let $X^\sharp$ and $Y^\sharp$ be arbitrary elements in $\mathcal{D}^\sharp$ and $s = (\delta_n, \theta_n)\cdots(\delta_1, \theta_1)$ be an arbitrary element in $\mathcal{S}$. By the definitions of $\gamma^\sharp$ and $\sqcap^\sharp$,

$$\begin{array}{rcl}
s \in \gamma^\sharp(X^\sharp \sqcap^\sharp Y^\sharp) & \Leftrightarrow & \left(\begin{array}{c} \forall 1 \le i \le n.(\theta_i \in (X^\sharp \sqcap^\sharp Y^\sharp)(\delta_i)) \\ \wedge \\ \forall 1 \le j < n.(\delta_j \in suf(\delta_{j+1})) \end{array}\right) \\
& \Leftrightarrow & \left(\begin{array}{c} \forall 1 \le i \le n.(\theta_i \in (X^\sharp(\delta_i) \cap Y^\sharp(\delta_i)) \\ \wedge \\ \forall 1 \le j < n.(\delta_j \in suf(\delta_{j+1})) \end{array}\right) \\
& \Leftrightarrow & \left(\begin{array}{c} \forall 1 \le i \le n.(\theta_i \in X^\sharp(\delta_i)) \\ \wedge \\ \forall 1 \le i \le n.(\theta_i \in Y^\sharp(\delta_i)) \\ \wedge \\ \forall 1 \le j < n.(\delta_j \in suf(\delta_{j+1})) \end{array}\right)
\end{array}$$

$$\Leftrightarrow\ (s \in \gamma^\sharp(X^\sharp)) \wedge (s \in \gamma^\sharp(Y^\sharp))$$
$$\Leftrightarrow\ s \in (\gamma^\sharp(X^\sharp) \cap \gamma^\sharp(Y^\sharp))$$

Therefore, $\gamma^\sharp(X^\sharp \sqcap^\sharp Y^\sharp) = \gamma^\sharp(X^\sharp) \cap \gamma^\sharp(Y^\sharp)$ since $X^\sharp$, $Y^\sharp$ and s are arbitrarily chosen and hence γ is a complete meet-morphism. This completes the proof of the lemma.

A.4. Proof of lemma 4.3

The operational semantics $[\![P]\!]$ is first characterised as the fixed-point of the following function.

$$F(X) \stackrel{def}{=} \bigcup_{0 \le \jmath \le 2} F_\jmath(X) \tag{38}$$

$$F_0(X) \stackrel{def}{=} \{(\iota, \theta) \mid \theta \in \Theta_\iota\} \tag{39}$$

$$F_1(X) \stackrel{def}{=} \left\{ (pq\delta', \sigma)(q\delta', \theta)s \;\middle|\; \begin{array}{c} p \leftharpoonup q \in \mathcal{E}^{call} \\ \wedge \\ (q\delta', \theta)s \in X \\ \wedge \\ \sigma = unify(\mathbb{A}(q), \theta, \mathbb{H}(p), \epsilon) \neq fail \end{array} \right\} \tag{40}$$

$$F_2(X) \stackrel{def}{=} \left\{ (pq\delta' p\text{-}\delta'', \sigma)s \;\middle|\; \begin{array}{c} p \leftharpoonup q \in \mathcal{E}^{ret} \\ \wedge \\ (q\delta' p\text{-}\delta'', \theta)(p\text{-}\delta'', \omega)s \in X \\ \wedge \\ \sigma = unify(\mathbb{H}(q), \theta, \mathbb{A}(p\text{-}), \omega) \neq fail \end{array} \right\} \tag{41}$$

The function F is monotonic on $< \wp(\mathcal{S}), \subseteq >$. It can be verified that $[\![P]\!] = lfp\ F$.

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

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

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

Let k not be a limit ordinal. Let $s' \in F \uparrow k$. There is $0 \le \jmath \le 2$ such that $s' \in F_\jmath(F \uparrow (k-1))$ by Eq. 38 and Eq. 2.

Let $\jmath = 0$. By Eq. 39, $s' = (p, \theta)$ and $\theta \in \Theta_\iota$. So, by Eq. 3 and Eq. 2, $s' \in \gamma^\sharp(F^\sharp \uparrow k)$.

Let $\jmath = 1$. By Eq. 40, $s' = (pq\delta', \sigma)(q\delta', \theta)s$ such that $p \leftharpoonup q \in \mathcal{E}^{call}$, $(q\delta', \theta)s \in F \uparrow (k-1)$ and $\sigma = unify(\mathbb{A}(q), \theta, \mathbb{H}(p), \epsilon) \neq fail$. By the

induction hypothesis, $(q\delta', \theta)s \in \gamma^\sharp(F^\sharp \uparrow (k-1))$. By Eq. 4 and Eq. 2 and the monotonicity of $F^\sharp$, $s' \in \gamma^\sharp(F^\sharp \uparrow k)$.

Let $\jmath = 2$. By Eq. 41, $s' = (pq\delta' p\text{-}\delta'', \sigma)s$ and there is an ω such that $p \leftharpoonup q \in \mathcal{E}^{ret}$ and $(q\delta' p\text{-}\delta'', \theta)(p\text{-}\delta'', \omega)s \in F \uparrow (k-1)$ and $\sigma = \mathit{unify}(\mathbb{H}(q), \theta, \mathbb{A}(p\text{-}), \omega) \neq \mathit{fail}$. By the induction hypothesis, $(q\delta' p\text{-}\delta'', \theta)(p\text{-}\delta'', \omega)s \in \gamma^\sharp(F^\sharp \uparrow (k-1))$. Therefore, $s' \in \gamma^\sharp(F^\sharp \uparrow k)$ by Eq. 5 and Eq. 2 and the monotonicity of $F^\sharp$.

Therefore, $F \uparrow k \subseteq \gamma^\sharp(F^\sharp \uparrow k)$ for any ordinal k. This completes the proof of the lemma.

A.5. Proof of lemma 5.1

Let $X^\flat$ and $Y^\flat$ be arbitrary elements in $\mathcal{D}^\flat$. By the definitions of $\gamma^\flat$ and $\sqcap^\flat$,

$$\begin{aligned}
\gamma^\flat(X^\flat \sqcap^\flat Y^\flat) &= \lambda\delta \in \Delta.\gamma((X^\flat \sqcap^\flat Y^\flat)(\beta(\delta))) \\
&= \lambda\delta \in \Delta.\gamma(X^\flat(\beta(\delta)) \sqcap Y^\flat(\beta(\delta))) \\
&= \lambda\delta \in \Delta.(\gamma(X^\flat(\beta(\delta))) \cap \gamma(Y^\flat(\beta(\delta)))) \\
&= \gamma^\flat(X^\flat) \cap \gamma^\flat(Y^\flat)
\end{aligned}$$

where the fourth formula is derived from the third using C2. Since $X^\flat$ and $Y^\flat$ are arbitrarily chosen, $\gamma^\flat$ is a complete meet-morphism. This completes the proof of the lemma.

A.6. Proof of theorem 5.2

The condition (C4) implies that $F^\flat$ is monotonic and therefore $\mathit{lfp}\ F^\flat$ exists. It suffices to prove that, for any $X^\flat \in \mathcal{D}^\flat$, $F^\sharp \circ \gamma^\flat(X^\flat) \sqsubseteq^\sharp \gamma^\flat \circ F^\flat(X^\flat)$. Let $\sigma \in [F^\sharp \circ \gamma^\flat(X^\flat)](\delta)$. It suffices to prove

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

for any $\delta \in \Delta$.

Let $\delta = \iota$. By Eq. 3, $\sigma \in \gamma(\pi_\iota)$. By Eq. 8, $\sigma \in \gamma([F^\flat(X^\flat)](\beta(\delta)))$. Thus, $\sigma \in [\gamma^\flat \circ F^\flat(X^\flat)](\delta)$ by Eq. 7.

Let $\delta = pq\delta'$ such that $p \leftharpoonup q \in \mathcal{E}^{call}$. By Eq. 4, $\sigma \in \mathit{cunify}(\mathbb{A}(q), [\gamma^\flat(X^\flat)](q\delta'), \mathbb{H}(p), \{\epsilon\})$. By Eq. 4 and Eq. 7, C3 and the monotonicity of function *cunify* in its fourth argument,

$$\begin{aligned}
\sigma &\in \mathit{cunify}(\mathbb{A}(q), \gamma(X^\flat(\beta(q\delta'))), \mathbb{H}(p), \gamma(\mathsf{id})) \\
&\subseteq \gamma \circ \mathit{aunify}(\mathbb{A}(q), X^\flat(\beta(q\delta')), \mathbb{H}(p), \mathsf{id})
\end{aligned}$$

So, by Eq. 7 and Eq. 9 and the monotonicity of γ,

$$\begin{array}{rcl} \sigma & \in & \gamma([F^\flat(X^\flat)](\beta(pq\delta')) \\ & \subseteq & \gamma([F^\flat(X^\flat)](\beta(pq\delta')) \\ & = & [\gamma^\flat \circ F^\flat(X^\flat)](pq\delta') \\ & = & [\gamma^\flat \circ F^\flat(X^\flat)](\delta) \end{array}$$

Let $\delta = pq\delta' p\text{-}\delta''$ such that $p \leftharpoonup q \in \mathcal{E}^{ret}$ and $q\delta' \in \Delta_c$. Then

$$\sigma \in \mathit{cunify}(\mathbb{H}(q), [\gamma^\flat(X^\flat)](q\delta' p\text{-}\delta''), \mathbb{A}(p\text{-}), [\gamma^\flat(X^\flat)](p\text{-}\delta''))$$

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

$$\begin{array}{rcl} \sigma & \in & \mathit{cunify}(\mathbb{H}(q), \gamma(X^\flat(\beta(q\delta' p\text{-}\delta''))), \mathbb{A}(p\text{-}), \gamma(X^\flat(\beta(p\text{-}\delta'')))) \\ & \subseteq & \gamma \circ \mathit{aunify}(\mathbb{H}(q), X^\flat(\beta(q\delta' p\text{-}\delta'')), \mathbb{A}(p\text{-}), X^\flat(\beta(p\text{-}\delta''))) \\ & \subseteq & \gamma([F^\flat(X^\flat)](\beta(\delta))) \\ & = & [\gamma^\flat \circ F^\flat(X^\flat)](\delta) \end{array}$$

This completes the proof of the theorem.