Inferring Precise Polymorphic Type Dependencies in Logic Programs

Lunjin Lu
Oakland University, MI 48309, USA

ABSTRACT

We present a new analysis that infers polymorphic type dependencies in logic programs. The analysis infers more precise information than previous type dependency inference analyses. The improvement in precision is achieved by making use of set union as a type constructor and non-deterministic type definitions.

Keywords: Abstract interpretation, Type dependency, Type inference

1. INTRODUCTION

In logic programming, types have been used in compile-time optimization, program transformation and error detection. A type inference analysis derives type information from the text of the program. Many type inference analyses have been proposed for logic programs. Most of those proposed analyses disregard dependencies between types of terms bound to different variables, which causes loss of precision. To the best of our knowledge, only analyses in [2, 6, 7, 14, 16] capture polymorphic type dependencies.

The analysis in [2] infers a set of abstract atoms $p(\tau_1, \ldots, \tau_n)$ where p is a predicate symbol and each τ_i is a poly-type - a type expression that may contain type parameters. Each abstract atom describes a set of atoms in the success set of the program. Let $\ell()$ be the list type constructor. Then, abstract atom $p(\beta, \ell(\beta))$ with β being a type parameter describes $p(t_1, t_2)$ iff there is a type R such that t_1 is of type R and t_2 is of type $\ell(R)$. For instance, the abstract atom describes $p(1, [2, 3])$ where the term 1 is of type *int* and the term $[2, 3]$ of type $\ell(int)$. The analysis in [6] is similar to that in [2] except that its inferred abstract atoms describe the whole success set of the program while the result of [2] describes only well-typed atoms in the success set. As an improvement on [6], an associative, commutative and idempotent operator $\oplus$ is introduced in [7] to form the type of a term from those of its sub-terms. Type inference analyses in [2, 6, 7] capture type dependency via type parameters as in the abstract atom $p(\beta, \ell(\beta))$. Hill and Spoto [14] provide a method that enriches an abstract domain with dependency information. The enriched domain contains elements like $(x \in \beta) \rightarrow (y \in \ell(\beta))$ meaning that if x is of type β then y is of type $\ell(\beta)$ for any type β. This kind of type dependencies are more explicit than those in [2, 6, 7]. The above mentioned analyses are goal-independent in the sense that they do not require a type description of initial goals as an analysis input. A goal-dependent analysis is performed for a given type description of initial goals. Given a type description of initial goals, the goal-dependent analysis in [16] infers a type description for each program point. Type descriptions are parameterized by type parameters that can be instantiated after analysis, i.e., it infers the dependency of the type description at each program point on that of initial goals.

Two factors compromise precision of the analyses in [2, 6, 7, 14, 16]. Firstly, they only allow deterministic type definitions in that a function symbol cannot occur in more than one definition for the same type constructor. A type then denotes a term language recognized by a deterministic top-down tree automaton [8] and hence it is called deterministic. This causes loss of precision because of the limited power of deterministic types. The same restriction also prevents many natural typings. For instance, these two type rules $float \multimap +(int, float)$ and $float \multimap +(float, float)$ violate the restriction. Secondly, the join operation on types in these analyses incurs a loss of precision. The denotation of the join of two types can be larger than the set union of their denotations. For instance, the join of $\ell(int)$ and $\ell(float)$ is $\ell(number)$. Let $\sqcup$ be a type constructor that is interpreted as set union. Then $\ell(number)$ is a super-type of $\sqcup(\ell(int), \ell(float))$ since the list [1, 0.5] belongs to the former but not the latter.

This paper presents a new analysis that infers polymorphic type dependencies. Following [2, 6, 7, 14, 16], the analysis is performed with *a priori* type definitions which determine possible types and their meanings. Types are formed from a fixed alphabet of type constructors. In contrast, a type analysis that infers both type definitions and types such as [5] has the freedom of generating new type constructors as needed. The main contribution of the work is the use of set union as a type constructor and non-deterministic type definitions. The set union as a type constructor allows us to compute precisely the join of two types. Non-deterministic type definitions enable us to use types to denote more sets of terms. They together make the new analysis more precise than those in [2, 6, 7, 14, 16].

The remainder of the paper is organized as follows. Section 2 recalls basic concepts in logic programming [15] and abstract interpretation [9] that are used in the rest of the paper, and reformulates the S-semantics [3] in terms of renaming, unification and projection operations on substitutions. The S-semantics will be used as the concrete semantics for the new analysis. Section 3 presents type definitions and the meanings of types. In section 4, we present abstract substitutions and renaming, unification and projection operations on abstract substitutions. Each abstract substitution describes a set of substitutions and each operation on abstract substitutions simulates an operation on substitutions. Section 5 presents the abstract semantics for the new analysis and illustrates it via examples. Section 6 concludes. Proofs are placed in an appendix.

2. PRELIMINARIES

We assume a set of function symbols Σ, a set of predicate symbols Π and an infinite set of variables $\mathcal{V}$. Let $V \subseteq \mathcal{V}$. We use $\mathsf{Term}(\Sigma, V)$ to denote the set of terms that are constructed from the function symbols in Σ and the variables in V. Let $\mathbf{V}(o)$ be the set of variables in a syntactic object o. The set of subsets of a set S is denoted by $\wp(S)$ and the set of finite subsets of S by $\wp_f(S)$. A bold lower case letter denotes a sequence of different variables. An atom is of the form $p(\vec{x})$ with $p \in \Pi$ and $\vec{x}$ a sequence of different variables with a suitable dimension. When there is no ambiguity, $\mathbf{V}(\vec{x})$ will be written as $\vec{x}$.

2.1 Abstract Interpretation

A semantics of a program is defined as the least fixpoint of a monotonic function on a complete lattice. There are two semantics in abstract interpretation: concrete and abstract semantics. Let *lfp* be the least fixpoint operator. The concrete semantics is *lfp* $\mathcal{C}$ where $\mathcal{C}$ is a monotone function on the concrete domain $\langle C, \sqsubseteq_C \rangle$ while the abstract semantics is *lfp* $\mathcal{A}$ where $\mathcal{A}$ is a monotone function on the abstract domain $\langle A, \sqsubseteq_A \rangle$. The two domains are related via a Galois connection (α, γ) which 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 γ a concretization 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 [10]. 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 $X \subseteq A$.

2.2 Equivalence on substitutions

A substitution θ is a mapping from $\mathcal{V}$ to $\mathsf{Term}(\Sigma, \mathcal{V})$ such that its domain $dom(\theta) = \{x \mid x \neq \theta(x)\}$ is finite. A substitution θ is idempotent iff $\theta(\theta(x)) = \theta(x)$ for all $x \in \mathcal{V}$. The set of all idempotent substitutions is denoted *Subst*. A renaming substitution is a bijection from $\mathcal{V}$ to $\mathcal{V}$. Two computed answer substitutions θ and σ for a goal G are equivalent iff there is a renaming substitution δ such that $\theta(x) = \delta(\sigma(x))$ for all $x \in \mathbf{V}(G)$ [15]. This notion of equivalence is formalized by the following relation $\sim_U$ that was introduced in [1]. Let $U \in \wp_f(\mathcal{V})$ and $\theta, \sigma \in \mathit{Subst}$. We write $\theta \leq_U \sigma$ iff there is a substitution δ such that $\theta(x) = \delta(\sigma(x))$ for all $x \in U$. For instance, let $\theta = \{x \mapsto f(a), y \mapsto f(b), z \mapsto h(a)\}$ and $\sigma = \{x \mapsto u, y \mapsto v, z \mapsto g(b)\}$ and $\delta = \{u \mapsto f(a), v \mapsto f(b)\}$. Then $\theta(x) = \delta(\sigma(x))$ and $\theta(y) = \delta(\sigma(y))$ and hence $\theta \leq_{\{x,y\}} \sigma$. The relation $\leq_U$ is a pre-order. Let $\sim_U$ be the equivalence relation induced by $\leq_U$. Then, $\theta \sim_U \sigma$ iff there is a renaming substitution ρ such that $\theta(x) = \rho(\sigma(x))$ for all $x \in U$. Note that $\sim_U$ is weaker than the usual equivalence relation in which θ and $\rho \circ \sigma$ must agree on all variables in $\mathcal{V}$.

Let $[\theta]_U$ denote the equivalence class of θ with respect to $\sim_U$ and Subst_U the quotient set of *Subst* with respect to $\sim_U$. A substitution θ' is a canonical representative of an equivalence class $[\theta]_U$ iff $\theta' \in [\theta]_U$ and $dom(\theta') = U$ and $U \cap rng(\theta') = \emptyset$ where $rng(\theta') = \cup_{x \in dom(\theta')} \mathbf{V}(\theta'(x))$. In any θ' in $[\theta]_U$, bindings for variables outside U is irrelevant since $\{x \mapsto \theta'(x) \mid x \in U\} \sim_U \theta'$. We will use η and ζ to denote equivalence classes of substitutions with respect to $\sim_U$ for some U that is either irrelevant or clear from the context. Define $\mathit{Subst}_\sim = \biguplus_{U \in \wp_f(\mathcal{V})} \mathit{Subst}_U$ where $\uplus$ is the disjoint union operation. Let $\diamond \notin \mathit{Subst}_\sim$ indicate failure of unification and define $\mathit{Subst}_\sim^\diamond = \mathit{Subst}_\sim \cup \{\diamond\}$. Elements in $\mathit{Subst}_\sim^\diamond$ are ordered by $\leq$ that is defined by $\diamond \leq e$ for all $e \in \mathit{Subst}_\sim^\diamond$ and for all $[\theta_1]_U, [\theta_2]_V \in \mathit{Subst}_\sim$,

$$[\theta_1]_U \leq [\theta_2]_V \textit{ iff } (U \supseteq V) \wedge (\theta_1 \leq_V \theta_2)$$

2.3 Operations on substitutions

An equational constraint is a finite set (conjunction) of equations of the form $t_1 = t_2$ with t_i for $i = 1, 2$ being terms. Define $mgu(E)$ as the $\sim_{\mathbf{V}(E)}$ equivalence class of most general unifiers for E if E is unifiable. Otherwise, $mgu(E) = \diamond$.

One operation performed during program execution is to conjoin constraints represented by substitutions. The unification operation $\odot : \mathit{Subst}_U \times \mathit{Subst}_V \mapsto \mathit{Subst}_{U \cup V} \cup \{\diamond\}$ is defined by $[\theta_1]_U \odot [\theta_2]_V = mgu(eq(\theta_1') \cup eq(\theta_2'))$ where $eq(\theta) = \{x = \theta(x) \mid x \in dom(\theta)\}$ and θ_1' and θ_2' are respectively canonical representatives of $[\theta_1]_U$ and $[\theta_2]_V$ such that $(U \cup \mathbf{V}(\theta_1')) \cap (V \cup \mathbf{V}(\theta_2')) \subseteq U \cap V$. Another operation is projection $\pi_X : \mathit{Subst}_U \mapsto \mathit{Subst}_{U \setminus X}$ for $X \in \wp_f(\mathcal{V})$ defined as $\pi_X([\theta]_U) = [\theta]_{U \setminus X}$. The operator π_X hides variables in X. A third operation is renaming defined as follows. If $\vec{x} \cap \vec{y} = \emptyset$ then $\mathcal{R}_{\vec{x} \mapsto \vec{y}}(\eta) = \pi_{\vec{x}}(mgu(\{\vec{x} = \vec{y}\}) \odot \eta)$. Otherwise, $\mathcal{R}_{\vec{x} \mapsto \vec{y}}(\eta) = \mathcal{R}_{\vec{z} \mapsto \vec{y}}(\mathcal{R}_{\vec{x} \mapsto \vec{z}}(\eta))$ where $\vec{z} \cap (\vec{x} \cup \vec{y}) = \emptyset$. The operation $\mathcal{R}_{\vec{x} \mapsto \vec{y}}(\cdot)$ transforms an equational constraint on $\vec{x}$ to one on $\vec{y}$.

2.4 S-Semantics

The S-semantics is a bottom-up and fixpoint definition of the set of computed answers of a program [3]. The computed answer for a predicate $p(\vec{x})$ is a set of ($\sim_{\vec{x}}$-equivalence classes of) substitutions. Given a computed answer for $p(\vec{x})$, one can obtain a computed answer for $p(\vec{y})$ via renaming. Thus, we reserve a set of special variables $\Lambda = \{\lambda_1, \cdots, \lambda_n\}$ where n is the maximum arity of the predicates in Π and define the meaning of a predicate p of arity m as the set of computed answers for $p(\lambda_1, \cdots, \lambda_m)$. We further denote the sequence of arguments in $p(\lambda_1, \cdots, \lambda_m)$ as $\Lambda(p)$.

An interpretation is a set of pairs $\langle p, \eta\rangle$ where p is a predicate and η is an element of $Subst_{\Lambda(p)}$. The domain of interpretations is

$$\mathsf{Int} = \wp(\{\langle p, \eta\rangle \mid p \in \Pi \text{ and } \eta \in Subst_{\Lambda(p)}\})$$

Interpretations are ordered by set inclusion $\subseteq$ and $\langle \mathsf{Int}, \subseteq\rangle$ is a complete lattice.

For the purpose of calculating computed answers, we can assume that a clause have the form $p(\vec{x}) \leftarrow E, A_1, \cdots, A_n$ where $n \geq 0$, E is an equational constraint and each A_i an atom. This is a direct consequence of the computational rule independence result [15]. We may further assume that the equational constraint E is unifiable for otherwise the clause is useless. The S-semantics of a program P is *lfp* T_P where $T_P : \mathsf{Int} \mapsto \mathsf{Int}$ is defined

$$T_P(I) = \left\{ \langle p, \eta\rangle \left| \begin{array}{l} p \in \Pi, \\ (p(\vec{x}_0) \leftarrow E, p_1(\vec{x}_1), \cdots, p_n(\vec{x}_n)) \in P, \\ \langle p_1, \eta_1\rangle \in I, \\ \cdots, \\ \langle p_n, \eta_n\rangle \in I, \\ \zeta_1 = \mathcal{R}_{\Lambda(p_1)\mapsto\vec{x}_1}(\eta_1), \\ \cdots, \\ \zeta_n = \mathcal{R}_{\Lambda(p_n)\mapsto\vec{x}_n}(\eta_n), \\ L = (\mathbf{V}(E) \cup \bigcup_{1\leq i\leq n} \vec{x}_i) \setminus \vec{x}_0, \\ \eta = \mathcal{R}_{\vec{x}_0\mapsto\Lambda(p)}(\pi_L(mgu(E) \odot \zeta_1 \odot \cdots \odot \zeta_n)) \end{array} \right. \right\}$$

The computed answers for an atom $p(\vec{x}_0)$ are computed from the computed answers for the bodies of its defining clauses. Consider a clause $p(\vec{x}_0) \leftarrow B$. If σ is a computed answer for B then $\pi_{\mathbf{V}(B)\setminus\vec{x}_0}(\sigma)$ is a computed answer for $p(\vec{x}_0)$. The computed answer for B is computed by conjoining those for the atoms in B and the most general unifier of the equational constraint in B through $\odot$.

3. TYPE SYSTEM

A type denotes a set of terms that is closed under instantiation [4]. A type is represented by a ground term constructed from type constructors in $\mathsf{Cons} \cup \{\sqcup, \mathbf{1}, \mathbf{0}\}$. Thus, a type is a term in $\mathsf{RT} = \mathsf{Term}(\mathsf{Cons} \cup \{\sqcup, \mathbf{1}, \mathbf{0}\}, \emptyset)$. It is assumed that $(\mathsf{Cons} \cup \{\sqcup, \mathbf{1}, \mathbf{0}\}) \cap \Sigma = \emptyset$.

We assume an infinite set of type parameters Para. Types are defined by type rules [11]. A type rule is of the form $c(\beta_1, \cdots, \beta_m) \twoheadrightarrow f(\tau_1, \cdots, \tau_n)$ where $c \in \mathsf{Cons}, f \in \Sigma$ and $\beta_1, \cdots, \beta_m$ are different type parameters, and each τ_j is either in $\{\beta_1, \cdots, \beta_m\}$ or of the form $d(\beta'_1, \cdots, \beta'_k)$ with $\beta'_1, \cdots, \beta'_k$ being different type parameters in $\{\beta_1, \cdots, \beta_m\}$ and $d \in \mathsf{Cons}$. A type parameter may appear in τ_i and τ_j for different $1 \leq i, j \leq n$. Every type parameter in the righthand side of a type rule must occur in its lefthand side. This restriction has been adopted in all type definition formalisms. Overloading of function symbols is permitted as a function symbol can appear in the righthand sides of multiple type rules. We assume that each function symbol in Σ occurs in at least one type rule. A poly-type is a term in $\mathsf{Term}(\mathsf{Cons} \cup \{\sqcup, \mathbf{1}, \mathbf{0}\}, \mathsf{Para})$ denoted Poly. Let $\mathsf{Para}(o)$ be the set of type parameters occurring in a syntactic object o.

EXAMPLE 3.1. *Let* $\Sigma = \{0, s(), [\ \ ], [\ \ |\ \ ]\}$ *and* $\mathsf{Cons} = \{nat, even, odd, \ell()\}$. *This set of type rules defines natural numbers, even numbers, odd numbers, and lists.*

$$\Delta = \left\{ \begin{array}{ll} nat \twoheadrightarrow 0, & nat \twoheadrightarrow s(nat), \\ even \twoheadrightarrow 0, & even \twoheadrightarrow s(odd), \\ odd \twoheadrightarrow s(even), & \\ \ell(\beta) \twoheadrightarrow [\ ], & \ell(\beta) \twoheadrightarrow [\beta|\ell(\beta)] \end{array} \right\}$$

A type substitution κ is a mapping from Para to Poly with a finite domain $dom(\kappa) = \{\beta \mid \kappa(\beta) \neq \beta\}$. The application of $\Bbbk$ to a syntactic object o is to replace each occurrence of β in o with $\Bbbk(\beta)$. For instance, $\{\beta_1 \mapsto \ell(nat), \beta_2 \mapsto nat\}(\ell(\beta_1)) = \ell(\ell(nat))$ and $\{\beta \mapsto nat\}(\ell(\beta) \twoheadrightarrow [\beta|\ell(\beta)] = (\ell(nat) \twoheadrightarrow [nat|\ell(nat)]$. The join $\kappa_1 \curlyvee \kappa_2$ of type substitutions κ_1 and κ_2 is defined

$$(\kappa_1 \curlyvee \kappa_2)(\beta) = \left\{ \begin{array}{ll} \kappa_1(\beta) \sqcup \kappa_2(\beta), & \beta \in (dom(\kappa_1) \cap dom(\kappa_2)) \\ \kappa_1(\beta), & \beta \in (dom(\kappa_1) \setminus dom(\kappa_2)) \\ \kappa_2(\beta), & \beta \in (dom(\kappa_2) \setminus dom(\kappa_1)) \\ \beta, & \beta \notin (dom(\kappa_1) \cup dom(\kappa_2)) \end{array} \right.$$

Note that $dom(\kappa_1 \curlyvee \kappa_2) = dom(\kappa_1) \cup dom(\kappa_2)$. A type valuation κ is a type substitution such that $\kappa(\beta) \in \mathsf{RT}$ for each $\beta \in dom(\kappa)$, i.e., κ maps each type parameter in its domain to a type. Let T_1 and T_2 be poly-types. Let $match(T_1, T_2) = \kappa$ if there is a type substitution κ such that $\kappa(T_1) = T_2$. Otherwise, $match(T_1, T_2) = \diamond$.

Given a set of type rules Δ, the meaning of a type is defined as follows.

$$\begin{array}{rcl} [\![\mathbf{1}]\!]_\Delta & = & \mathsf{Term} \\ [\![\mathbf{0}]\!]_\Delta & = & \emptyset \\ [\![R_1 \sqcup R_2]\!]_\Delta & = & [\![R_1]\!]_\Delta \cup [\![R_2]\!]_\Delta \\ [\![c(R_1, \cdots, R_m)]\!]_\Delta & = & \end{array}$$
$$\left\{ f(t_1, \cdots, t_n) \left| \begin{array}{l} (c(\beta_1, \cdots, \beta_m) \twoheadrightarrow f(\tau_1, \cdots, \tau_n)) \in \Delta, \\ \kappa = \{\beta_j \mapsto R_j \mid 1 \leq j \leq m\}, \\ \forall 1 \leq i \leq n.(\ t_i \in [\![\kappa(\tau_i)]\!]_\Delta) \end{array} \right. \right\}$$

The semantic function $[\![\cdot]\!]_\Delta$ interprets the type constructor $\sqcup$ as set union, $\mathbf{1}$ as Term and $\mathbf{0}$ as the empty set. Every type in RT denotes a regular term language [17]. Furthermore, types are closed under instantiation in that if $t \in [\![R]\!]_\Delta$ then $\sigma(t) \in [\![R]\!]_\Delta$ for any term t, type R, and substitution σ [17].

EXAMPLE 3.2. *Let* Δ *be that in example 3.1. Then*

$$\begin{array}{rcl} [\![nat]\!]_\Delta & = & \{0, s(0), s(s(0)), \cdots\} \\ [\![\ell(\mathbf{0})]\!]_\Delta & = & \{[\ ]\} \\ [\![\ell(\mathbf{1})]\!]_\Delta & = & \{[\ ], [x], \cdots\} \ \ \textit{where } x \in \mathcal{V} \end{array}$$

4. ABSTRACT SUBSTITUTIONS

During analysis, substitutions are described by abstract substitutions formed of type conditions $x \in T$ with $x \in \mathcal{V}$ and $T \in \mathsf{Poly}$, and connectives $\wedge$ and $\Leftarrow$.

DEFINITION 4.1. *A type dependency is a formula of the form* $(x_0 \in T_0) \Leftarrow (x_1 \in T_1) \wedge \cdots \wedge (x_m \in T_m)$ *such that*

- $\mathsf{Para}(T_j) \subseteq \mathsf{Para}(T_0)$ *for all* $1 \leq j \leq m$;
- T_j *contains no occurrence of* $\sqcup$ *for all* $1 \leq j \leq m$ *and*

- $\mathsf{Para}(T_i) \cap \mathsf{Para}(T_j) = \emptyset$ *for all* $1 \leq i, j \leq m$ *such that* $i \neq j$.

A type dependency is normalized if $x_i \neq x_j$ *for all* $0 \leq i, j \leq m$ *such that* $i \neq j$. *The above restrictions on the form of a normalized type dependency are preserved by all analysis operations defined later. If* $m = 0$ *then the above type dependency is written as* $(x_0 \in T_0) \Leftarrow \epsilon$ *or simply* $(x_0 \in T_0)$.

An abstract substitution is a set of normalized type dependencies. An abstract substitution describes a set of substitutions. For instance, the abstract substitution $y \in \ell(nat) \Leftarrow x \in \ell(nat)$ describes all those substitutions θ such that if $\theta(x)$ is a list of natural numbers then $\theta(y)$ is a list of natural numbers. This is formalized by the following satisfiability relation $\models$.

- $\theta \models (x \in R)$ iff $\theta(x) \in [R]_\triangle$ where $x \in \mathcal{V}$ and $R \in \mathsf{RT}$.
- $\theta \models ((x_0 \in T_0) \Leftarrow (x_1 \in T_1) \wedge \cdots \wedge (x_m \in T_m))$ iff, for each type valuation κ such that $dom(\kappa) \supseteq \cup_{0 \leq j \leq m} \mathsf{Para}(T_j)$, either $\theta \models (x_0 \in \kappa(T_0))$ or $\theta \not\models (x_i \in \kappa(T_i))$ for some $1 \leq i \leq m$;
- $\theta \models \{c_1, \cdots, c_n\}$ iff $\theta \models c_i$ for all $1 \leq i \leq n$.

Note that each type valuation in the definition of satisfiability of a type dependency instantiates each poly-type in the type dependency into a type. We write $\phi_1 \preccurlyeq \phi_2$ iff $\theta \models \phi_1$ implies $\theta \models \phi_2$ for every substitution θ and $\phi_1 \approx \phi_2$ iff $\phi_1 \preccurlyeq \phi_2$ and $\phi_2 \preccurlyeq \phi_1$. We say that ϕ_1 and ϕ_2 are equivalent if $\phi_1 \approx \phi_2$. It follows that renaming of type parameters in a type dependency results in an equivalent type dependency. That is, $\{c\} \approx \{c[\beta'/\beta]\}$ where β' is a type parameter that does not occur in c and $c[\beta'/\beta]$ is obtained from replacing each occurrence of β in c with β'. Let $[\phi]_\approx$ denote the equivalence class of $\approx$ that contains ϕ. Define $[\phi_1]_\approx \leq [\phi_2]_\approx$ iff $\phi_1 \preccurlyeq \phi_2$. Then $\leq$ is a partial order on equivalence classes of $\approx$. Define $\theta \models [\phi]_\approx$ iff $\theta \models \phi$. Let U be a set of variables. An abstract substitution ϕ on U is such that all variables in ϕ are contained in U. Let TDC_U be the set of $\approx$-equivalence classes of abstract substitutions on U.

There are type dependencies that do not contain useful information in that they are satisfied by all substitutions. Such type dependencies are called tautologies. Formally, a type dependency c is a tautology iff $\theta \models c$ for any substitution θ. Let c be $(xs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow xs \in \ell(\beta_1))$. Let θ be an arbitrary substitution and R_1 and R_2 be arbitrary types. If $\theta \not\models (xs \in \ell(R_1))$ then $\theta \models (xs \in \ell(R_1 \sqcup R_2) \Leftarrow xs \in \ell(R_1))$ by the definition of $\models$. Otherwise, $\theta \models (xs \in \ell(R_1 \sqcup R_1))$ and hence $\theta \models (xs \in \ell(R_1 \sqcup R_2) \Leftarrow xs \in \ell(R_1))$ since $[\ell(R_1)]_\triangle \subseteq [\ell(R_1 \sqcup R_2)]_\triangle$. Therefore, $\theta \models c$ for any substitution θ and hence c is a tautology. Removal of tautologies from an abstract substitution results in an equivalent abstract substitution.

4.1 Operations on abstract substitutions

During analysis, operations on substitutions are simulated by operations on abstract substitutions. We first define the operations on abstract substitutions and then present the correctness of these operations.

4.1.1 Abstract Unification

The unification operation $\odot$ is simulated by an abstract unification operation $\odot^\sharp : \mathsf{TDC}_U \times \mathsf{TDC}_V \mapsto \mathsf{TDC}_{U \cup V}$ defined as $\phi_1 \odot^\sharp \phi_2 = \phi_1 \cup \phi_2$.

4.1.2 Abstract Projection

The concrete projection operation π_X is simulated by an abstract operation $\pi^\sharp_X$ which is defined in terms of an unfolding operation which is in turn defined in terms of a normalization operation. The normalization operation takes a type dependency whose body contains at most two type conditions for each variable and normalizes it. The unfolding operation unfolds a normalized type dependency with another to obtain a new normalized type dependency. The normalization operation is defined

$$\begin{array}{rcl} norm(h \Leftarrow \epsilon) & = & (h \Leftarrow \epsilon) \\ norm(h \Leftarrow (x \in T \wedge \mu)) & = & fold(x \in T, norm(h \Leftarrow \mu)) \end{array}$$

where

$$fold(x \in T_1, y \in T \Leftarrow \mu)$$
$$= \left\{ \begin{array}{ll} (y \in T \Leftarrow (x \in T_1 \wedge \mu)), & \text{If } x \notin \mathbf{V}(\mu), \\ (y \in \kappa(T) \Leftarrow \mu_1 \wedge x \in T_2 \wedge \mu_2), & \\ \quad \text{If} \left(\begin{array}{l} \mu = (\mu_1 \wedge x \in T_2 \wedge \mu_2) \\ \kappa = match(T_1, T_2) \neq \diamond \end{array} \right), & \\ (y \in \kappa(T) \Leftarrow \mu_1 \wedge x \in T_1 \wedge \mu_2), & \\ \quad \text{If} \left(\begin{array}{l} \mu = (\mu_1 \wedge x \in T_2 \wedge \mu_2) \\ \kappa = match(T_2, T_1) \neq \diamond \end{array} \right), & \\ (y \in \mathbf{1}), & Otherwise. \end{array} \right.$$

The operation $fold(x \in T_1, y \in T \Leftarrow \mu)$ adds $x \in T_1$ to μ if x does not occur in μ. Otherwise, μ contains $x \in T_2$ for some T_2. If there is $\kappa \neq \diamond$ such that either $\kappa = match(T_1, T_2)$ or $\kappa = match(T_2, T_1)$ then κ is applied to T and x is constrained by T_2 in the former case and by T_1 in the latter case. Otherwise, $(y \in T \Leftarrow x \in T_1 \wedge \mu)$ is approximated by $y \in \mathbf{1}$.

EXAMPLE 4.2. *$norm(zs \in \ell(\beta_1 \sqcup \beta_2 \sqcup \beta_3) \Leftarrow xs \in \ell(\beta_3) \wedge xs \in \ell(\beta_1) \wedge ys \in \ell(\beta_2)) = zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow xs \in \ell(\beta_1) \wedge ys \in \ell(\beta_2)$.*

Unfolding. Let $c_1 = (y \in T \Leftarrow \mu_1 \wedge (x \in T_1) \wedge \mu_2)$ and $c_2 = ((x \in T_2) \Leftarrow \nu)$ be normalized type dependencies such that they do not share any type parameter. Assume that y does not occur in ν. If there is a type substitution κ such that $\kappa_1(T_1) = T_2$, the unfolding of c_1 with c_2 is defined as $unf(c_1, c_2) = norm(y \in \kappa_1(T) \Leftarrow \mu_1 \wedge \nu \wedge \mu_2)$. If there is a type substitution κ_2 such that $\kappa_2(T_2) = T_1$, then the unfolding of c_1 with c_2 is defined as $unf(c_1, c_2) = norm(y \in T \Leftarrow \mu_1 \wedge \kappa_2(\nu) \wedge \mu_2)$. Otherwise, the unfolding of c_1 with c_2 is defined as $(y \in \mathbf{1})$ which is a tautology. Note that κ_i for $i = 1, 2$ are not applied to μ_1 or μ_2 since they do not share any type parameter with T_1, κ_1 is not applied to ν since it does not bind any type parameter in T_2, κ_2 is not applied to T since it does not bind any type parameter in T_1. If c_1 and c_2 share type parameters, then unfolding of c_1 with c_2 is defined as the unfolding of c_1 and c_2' where $c_2' \approx c_2$ such that c_1 and c_2 do not share type parameters. The unfolding operation is formally defined as follows.

$$unf(c_1, c_2) =$$
$$\left\{\begin{array}{l} let \left(\begin{array}{l} c_2' \approx c_2 \text{ s.t. } \mathsf{Para}(c_2) \cap \mathsf{Para}(c_2') = \emptyset \\ c_1 = (y \in T \Leftarrow \mu_1 \wedge (x \in T_1) \wedge \mu_2) \\ c_2' = (x \in T_2 \Leftarrow \nu) \\ \kappa_1 = match(T_1, T_2), \\ \kappa_2 = match(T_2, T_1) \end{array}\right) in \\ \left\{\begin{array}{ll} (y \in \mathbf{1}), & \text{If } y \in \mathbf{V}(\nu) \text{ or } \kappa_i = \diamond \text{ for all } i \in \{1,2\} \\ norm(y \in \kappa_1(T) \Leftarrow \mu_1 \wedge \nu \wedge \mu_2), & \text{If } \kappa_1 \neq \diamond \\ norm(y \in T \Leftarrow \mu_1 \wedge \kappa_2(\nu) \wedge \mu_2), & \text{Otherwise.} \end{array}\right. \end{array}\right.$$

Note that $unf(c_1, c_2)$ is normalized since both c_1 and c_2 are. If $\kappa_1 \neq \diamond$ and $\kappa_2 \neq \diamond$ then T_1 and T_2 are renaming variants and the last two cases of the above definition yield the same result.

EXAMPLE 4.3. *Let $c_1 = (zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow xs \in \ell(\beta_1) \wedge zs1 \in \ell(\beta_2))$ and $c_2 = (zs1 \in \ell(\beta_1 \sqcup nat) \Leftarrow ys \in \ell(\beta_1))$. We have $c_2 \approx c_2'$ where $c_2' = (zs1 \in \ell(\beta \sqcup nat) \Leftarrow ys \in \ell(\beta))$. The type condition $(zs1 \in \ell(\beta_2))$ occurs in the body of c_1 and the type condition $(zs1 \in \ell(\beta \sqcup nat))$ in the head of c_2'. Then $\kappa_1 = match(\ell(\beta_2), \ell(\beta \sqcup nat)) = \{\beta_2 \mapsto (\beta \sqcup nat)\}$ and hence $unf(c_1, c_2) = unf(c_1, c_2') = (zs \in \ell(\beta_1 \sqcup \beta \sqcup nat) \Leftarrow xs \in \ell(\beta_1) \wedge ys \in \ell(\beta))$.*

PROPOSITION 4.4. *Let $c_1 = (y \in T \Leftarrow \mu_1 \wedge (x \in T_1) \wedge \mu_2)$ and $c_2 = (x \in T_2 \Leftarrow \nu)$ be normalized type dependencies. For any substitution $\eta \in Subst_{\sim}$, if $\eta \models c_1$ and $\eta \models c_2$ then $\eta \models unf(c_1, c_2)$.*

Abstract Projection. To project out a variable x in an abstract substitution ϕ. Each of those type dependencies c_1 in ϕ whose bodies contain x is unfolded with each of those type dependencies c_2 in ϕ whose heads contain x. Define predicates $\mathcal{H}_x$ and $\mathcal{B}_x$ by $\mathcal{H}_x(c) = true$ iff x occurs in the head of c and $\mathcal{B}_x(c) = true$ iff x occurs in the body of c. Let $\psi[\![-x]\!]$ be the resultant from removing from ψ all the type dependencies c such that $\mathcal{H}_x(c)$ is *true* and then removing all the type conditions on x from the remaining type dependencies in ψ. The abstract projection operation $\pi^\sharp_X : \mathsf{TDC}_U \mapsto \mathsf{TDC}_{U \setminus X}$ is defined

$$\pi^\sharp_{\{x_1, \cdots, x_n\}}(\phi) = \pi^\sharp_{x_1} \circ \cdots \pi^\sharp_{x_n}(\phi)$$

where $\pi^\sharp_x : \mathsf{TDC}_U \mapsto \mathsf{TDC}_{U \setminus \{x\}}$ for a single variable x is defined $\pi^\sharp_x(\phi) = (\phi \cup \{unf(c_1, c_2) \mid c_1 \in \phi^x_1 \text{ and } c_2 \in \phi^x_2\})[\![-x]\!]$ with $\phi^x_1 = \{c \in \phi \mid \mathcal{B}_x(c)\}$ and $\phi^x_2 = \{c \in \phi \mid \mathcal{H}_x(c)\} \cup \{(x \in \mathbf{1})\}$. Unfolding is applied by $\pi^\sharp_x$ to propagate type dependency information before it eliminates the variable x.

EXAMPLE 4.5. *Let $\phi = \{c_1, c_2, c_3\}$ where $c_1 = (xs \in \ell(\beta) \Leftarrow x \in \beta)$, $c_2 = (x \in \beta \Leftarrow ys \in \ell(\beta))$ and $c_3 = (zs \in \ell(\beta_1 \sqcup \beta_2 \sqcup \beta_3) \Leftarrow x \in \beta_1 \wedge xs \in \ell(\beta_2) \wedge ys \in \ell(\beta_3))$. Then $\phi^x_1 = \{c_1, c_3\}$ and $\phi^x_2 = \{c_2, (x \in \mathbf{1})\}$. We calculate $unf(c_1, c_2) = (xs \in \ell(\beta) \Leftarrow ys \in \ell(\beta))$ and $unf(c_1, (x \in \mathbf{1})) = (xs \in \ell(\mathbf{1}))$ and $unf(c_3, (x \in \mathbf{1})) = (zs \in \ell(\mathbf{1} \sqcup \beta_2 \sqcup \beta_3) \Leftarrow xs \in \ell(\beta_2) \wedge ys \in \ell(\beta_3))$ which is equivalent to $(zs \in \ell(\mathbf{1}) \Leftarrow xs \in \ell(\beta_1) \wedge ys \in \ell(\beta_2))$. We continue to compute*

$$\begin{array}{rl} & unf(c_3, c_2)) \\ = & norm\left(zs \in \ell(\beta_1 \sqcup \beta_2 \sqcup \beta_3) \Leftarrow \left(\begin{array}{c} ys \in \ell(\beta_1) \\ \wedge \\ xs \in \ell(\beta_2) \\ \wedge \\ ys \in \ell(\beta_3) \end{array}\right)\right) \\ = & (zs \in \ell(\beta_1 \sqcup \beta_2 \sqcup \beta_1) \Leftarrow ys \in \ell(\beta_1) \wedge xs \in \ell(\beta_2)) \end{array}$$

which is equivalent to $(zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow xs \in \ell(\beta_1) \wedge ys \in \ell(\beta_2))$. Note also that type dependency $(zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow xs \in \ell(\beta_1) \wedge ys \in \ell(\beta_2))$ implies type dependency $(zs \in \ell(\mathbf{1}) \Leftarrow xs \in \ell(\beta_1) \wedge ys \in \ell(\beta_2))$. Therefore,

$$\pi^\sharp_x(\phi) = \left\{\begin{array}{c} (xs \in \ell(\beta) \Leftarrow ys \in \ell(\beta)),\ (xs \in \ell(\mathbf{1})), \\ (zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow xs \in \ell(\beta_1) \wedge ys \in \ell(\beta_2)) \end{array}\right\}$$

4.1.3 Abstract Renaming

The renaming operation $\mathcal{R}_{\vec{x} \mapsto \vec{y}}(\cdot)$ is simulated by an abstract renaming operation $\mathcal{R}^\sharp_{\vec{x} \mapsto \vec{y}}(\cdot)$. If $\vec{x} \cap \vec{y} = \emptyset$ then $\mathcal{R}^\sharp_{\vec{x} \mapsto \vec{y}}(\phi) = \phi[\vec{y}/\vec{x}]$. Otherwise, $\mathcal{R}^\sharp_{\vec{x} \mapsto \vec{y}}(\phi) = \mathcal{R}^\sharp_{\vec{z} \mapsto \vec{y}}(\mathcal{R}^\sharp_{\vec{x} \mapsto \vec{z}}(\phi))$ where $\vec{z} \cap (\vec{x} \cup \vec{y}) = \emptyset$.

The following theorem states that the abstract unification, projection and renaming operations simulate correctly corresponding concrete operations.

THEOREM 4.6. *If $\eta \models \phi$ and $\zeta \models \psi$ then (a) $(\eta \odot \zeta) \models (\phi \odot^\sharp \psi)$; (b) $\pi_X(\eta) \models \pi^\sharp_X(\phi)$; and (c) $\mathcal{R}_{\vec{x} \mapsto \vec{y}}(\eta) \models \mathcal{R}^\sharp_{\vec{x} \mapsto \vec{y}}(\phi)$.*

5. ABSTRACT SEMANTICS

The following append program and type definitions are used for illustration throughout the section.

```
a(xs,ys,zs) :- xs=[], ys=zs.              (C1)
a(xs,ys,zs) :- xs=[x|xs1], zs=[x|zs1],
               a(xs1,ys,zs1).             (C2)
```

$$\Delta = \{\ell(\beta) \twoheadrightarrow [\,],\ \ell(\beta) \twoheadrightarrow [\beta|\ell(\beta)]\}$$

The clauses in the program will be referred to as C1 and C2.

5.1 Abstraction of Equational Constraints

A key step in analysis is to abstract an equational constraint to a set of type dependencies. The abstraction of an equational constraint is derived from type rules. Let $\langle \tau_1, \cdots \tau_n \rangle$ be a sequence of poly-types. We use $ren(\tau_1, \cdots \tau_n)$ to denote a sequences of renaming type substitutions $\langle \kappa_1, \cdots, \kappa_n \rangle$ such that $dom(\kappa_i) = \mathsf{Para}(\tau_i)$ and $\mathsf{Para}(\kappa_i(\tau_i)) \cap \mathsf{Para}(\kappa_j(\tau_j)) = \emptyset$ for all $1 \leq i, j \leq n$ such that $i \neq j$. The poly-types are renamed apart from each other by the renaming type substitutions. The abstraction function α_Δ to be defined approximates polymorphic type dependencies among variables in equations. After unification, the two sides of an equation are instantiated into the same term and therefore have the same type. In the simpler case $y = x$, both x and y have the same type after unification but there is no constraint on the type. This fact is expressed by the abstract substitution $\{(y \in \beta \Leftarrow x \in \beta), (x \in \beta \Leftarrow y \in \beta)\}$.

EXAMPLE 5.1. *Consider the equation* `ys = zs` *in the clause* `C1`. *Its abstraction is* $\{(ys \in \beta \Leftarrow zs \in \beta), (zs \in \beta \Leftarrow ys \in \beta)\}$.

Now consider the more complex case $y = f(x_1, \cdots, x_n)$. The unification propagates type information from y to $x_1, \ldots, x_n$ and vice versa. Each type definition rule δ for f/n gives rise to two sets of type dependencies $\mathit{lfrt}(y, f(x_1, \cdots, x_n), \delta)$ and $\mathit{rtlf}(y, f(x_1, \cdots, x_n), \delta)$. The set $\mathit{lfrt}(y, f(x_1, \cdots, x_n), \delta)$ describes type propagation from y to $x_1, \ldots, x_n$ while $\mathit{rtlf}(y, f(x_1, \cdots, x_n), \delta)$ describes type propagation from x_1, $\ldots$, x_n to y. The former is defined

$$\begin{array}{c} \mathit{lfrt}(y, f(x_1, \cdots, x_n), \tau \twoheadrightarrow f(\tau_1, \cdots, \tau_n)) = \\ \{x_i \in \tau_i \Leftarrow y \in \tau \mid 1 \leq i \leq n\} \end{array}$$

EXAMPLE 5.2. *We have* $\mathit{lfrt}($`xs`, `[x/xs1]`$, \ell(\beta) \twoheadrightarrow [\beta|\ell(\beta)]) = \{(x \in \beta \Leftarrow xs \in \ell(\beta)), (xs1 \in \ell(\beta) \Leftarrow xs \in \ell(\beta))\}$ *and* $\mathit{lfrt}($`zs`, `[x/zs1]`$, \ell(\beta) \twoheadrightarrow [\beta|\ell(\beta)]) = \{(x \in \beta \Leftarrow zs \in \ell(\beta)), (zs1 \in \ell(\beta) \Leftarrow zs \in \ell(\beta))\}$.

Type propagation from $x_1, \ldots, x_n$ to y is more complicated. For instance, if h is of type *int* and t of type $\ell(\ell(\mathit{int}))$ then the type of $[h|t]$ is $\ell(\mathit{int} \sqcup \ell(\mathit{int}))$. Thus, after unification $l = [h|t]$ succeeds, l is of type $\ell(\mathit{int} \sqcup \ell(\mathit{int}))$. Without set union as a type constructor, the type of l would have to be approximated by $\ell(\mathbf{1})$ that does not capture information on the type of elements in the list. Type propagation from $x_1, \cdots, x_n$ to y is described by the following singleton set.

$$\begin{array}{l} \mathit{rtlf}(y, f(x_1, \cdots, x_n), \tau \twoheadrightarrow f(\tau_1, \cdots, \tau_n)) = \\ \left\{ \left(\begin{array}{l} \textbf{let } \langle \kappa_1, \cdots, \kappa_n \rangle = \mathit{ren}(\langle \tau_1, \cdots, \tau_n \rangle) \\ \quad\;\; \kappa = \kappa_1 \curlyvee \cdots \curlyvee \kappa_n \\ \textbf{in } \; (y \in \kappa(\tau) \Leftarrow \bigwedge_{i=1}^{n}(x_i \in \kappa_i(\tau_i))) \end{array} \right) \right\} \end{array}$$

EXAMPLE 5.3. *We have that*

$$\begin{array}{ll} \mathit{rtlf}(\texttt{xs}, [\,], \ell(\beta) \twoheadrightarrow [\,]) & = \quad \{(xs \in \ell(\mathbf{0}))\} \\ \mathit{rtlf}(\texttt{xs}, \texttt{[x/xs1]}, \ell(\beta) \twoheadrightarrow [\beta|\ell(\beta)]) & = \\ \quad \{(xs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge xs1 \in \ell(\beta_2))\} & \\ \mathit{rtlf}(\texttt{zs}, \texttt{[x/zs1]}, \ell(\beta) \twoheadrightarrow [\beta|\ell(\beta)]) & = \\ \quad \{(zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge zs1 \in \ell(\beta_2))\} & \end{array}$$

DEFINITION 5.4. *The abstraction of an equational constraint is defined* $\alpha_\Delta(E) = \bigcup_{e \in E}\{\alpha_\Delta(e)\}$ *where the abstraction of an equation is defined*

$$\begin{array}{rcl} \alpha_\Delta(y = x) & = & \left\{ \begin{array}{l} (y \in \beta \Leftarrow x \in \beta), \\ (x \in \beta \Leftarrow y \in \beta) \end{array} \right\} \\ \alpha_\Delta(y = f(x_1, \cdots, x_n)) & = & \\ & & \bigcup_{\delta \in \Delta_{f/n}} \left\{ \begin{array}{l} \mathit{lfrt}(y, f(x_1, \cdots, x_n), \delta), \\ \mathit{rtlf}(y, f(x_1, \cdots, x_n), \delta) \end{array} \right\} \end{array}$$

and $\Delta_{f/n}$ *is the set of all the type rules in* Δ *that are of the form* $\tau \twoheadrightarrow f(\tau_1, \cdots, \tau_n)$.

For a clause C of the form $p(\vec{x}_0) \leftarrow E, p_1(\vec{x}_1), \cdots, p_n(\vec{x}_n)$, let χ_C be $\alpha_\Delta(E)$.

EXAMPLE 5.5. *We have that* $\chi_{\texttt{C1}} = \alpha_\Delta(\texttt{xs} = [\,], \texttt{ys} = \texttt{zs}) = \{(xs \in \ell(\mathbf{0})), (ys \in \beta \Leftarrow zs \in \beta), (zs \in \beta \Leftarrow ys \in \beta)\}$ *and* $\chi_{\texttt{C2}} = \alpha_\Delta($`xs=[x/xs1], zs=[x/zs1]`$) = \{(x \in \beta \Leftarrow xs \in \ell(\beta)), (xs1 \in \ell(\beta) \Leftarrow xs \in \ell(\beta)), (xs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge xs1 \in \ell(\beta_2)), (x \in \beta \Leftarrow zs \in \ell(\beta)), (zs1 \in \ell(\beta) \Leftarrow zs \in \ell(\beta)), (zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge zs1 \in \ell(\beta_2))\}$.

5.2 Abstract Interpretations

The proposed analysis infers a set of success patterns that are pairs consisting of a predicate p and an abstract substitution $\phi \in \mathsf{TDC}_{\Lambda(p)}$. We say a success pattern $\langle p, \phi \rangle$ is subsumed by another $\langle p, \psi \rangle$ iff $\phi \leq \psi$. Two different sets of success patterns may contain the same information. For instance $\{\langle p, \{\lambda_1 \in \ell(\mathbf{1})\}\rangle\}$ contains the same information as $\{\langle p, \{\lambda_1 \in \ell(\mathbf{1})\}\rangle, \langle p, \{\lambda_1 \in \ell(\mathit{nat})\}\rangle\}$ since $\langle p, \{\lambda_1 \in \ell(\mathit{nat})\}\rangle$ is subsumed by $\langle p, \{\lambda_1 \in \ell(\mathbf{1})\}\rangle$. Let $\Phi \subseteq \{\langle p, \phi \rangle \mid p \in \Pi \text{ and } \phi \in \mathsf{TDC}_{\Lambda(p)}\}$ and define $\downarrow(\Phi) = \{\langle p, \psi \rangle \mid \langle p, \phi \rangle \in \Phi \text{ and } \psi \leq \phi\}$. The domain of abstract interpretations is

$$\mathsf{Int}^\sharp = \{\downarrow(I^\sharp) \mid I^\sharp \in \wp(\{\langle p, \phi \rangle \mid p \in \Pi \text{ and } \phi \in \mathsf{TDC}_{\Lambda(p)}\})\}$$

Note that $\langle \mathsf{Int}^\sharp, \subseteq \rangle$ is a complete lattice and an abstract interpretation can be represented by the collection of its success patterns that are not subsumed by other success patterns in it. In the sequel, we will omit the operator $\downarrow$ and simply write $\downarrow I^\sharp$ as $I^\sharp$ since both $I^\sharp$ and $\downarrow I^\sharp$ describe the same set of computed answers. A connection between Int and $\mathsf{Int}^\sharp$ is characterized by $\gamma : \mathsf{Int}^\sharp \mapsto \mathsf{Int}$ defined as

$$\gamma(I^\sharp) \quad = \quad \{\langle p, \eta \rangle \mid p \in \Pi \text{ and } \langle p, \phi \rangle \in I^\sharp \text{ and } \eta \models \phi\}$$

LEMMA 5.6. *The function* γ *is a complete meet-morphism.*

5.3 Abstract Semantics

The abstract semantic of a program P is $\mathit{lfp}\ T_P^\sharp$ where $T_P^\sharp : \mathsf{Int}^\sharp \mapsto \mathsf{Int}^\sharp$ is defined

$$T_P^\sharp(I^\sharp) = \left\{ \langle p, \phi \rangle \left| \begin{array}{l} p \in \Pi, \\ (p(\vec{x}_0) \leftarrow E, p_1(\vec{x}_1), \cdots, p_n(\vec{x}_n)) \in P, \\ \langle p_1, \phi_1 \rangle \in I^\sharp, \ldots, \langle p_n, \phi_n \rangle \in I^\sharp, \\ \psi_1 = \mathcal{R}^\sharp_{\Lambda(p_1) \mapsto \vec{x}_1}(\phi_1), \\ \cdots, \\ \psi_n = \mathcal{R}^\sharp_{\Lambda(p_n) \mapsto \vec{x}_n}\phi_n, \\ L = (\mathbf{V}(E) \cup \bigcup_{1 \leq i \leq n} \vec{x}_i) \setminus \vec{x}_0, \\ \phi = \mathcal{R}^\sharp_{\vec{x}_0 \mapsto \Lambda(p)} \circ \pi^\sharp_L(\alpha_\Delta(E) \odot^\sharp \psi_1 \odot^\sharp \cdots \odot^\sharp \psi_n) \end{array} \right. \right\}$$

Note that $\alpha_\Delta(E)$ can be computed before the least fixpoint computation. The abstract domain $\mathsf{Int}^\sharp$ is infinite, which may lead to non-termination of analysis. Termination can be achieved by limiting the depth of poly-types in an abstract substitution as is done in [14].

THEOREM 5.7. *The abstract semantics approximates the concrete semantics correctly, i.e.,* $(\mathit{lfp}\ T_P) \subseteq \gamma(\mathit{lfp}\ T_P^\sharp)$.

5.4 Analysis of Append

We now illustrate the analysis with the append program. The least fixpoint computation generates a series of iterates $I_0^\sharp, \cdots, I_i^\sharp, I_{i+1}^\sharp$ until $I_i^\sharp = I_{i+1}^\sharp$. The initial iterate is $I_0^\sharp = \{\}$ that describes the empty set of computed answers. Each later iterate $I_{i+1}^\sharp$ is computed as $T_P^\sharp(I_i^\sharp)$. Let $\vec{\lambda} = (\lambda_1, \lambda_2, \lambda_3)$, $\vec{v} = (\texttt{xs,ys,zs})$ and $\vec{v}' = (\texttt{xs1,ys,zs1})$.

Calculating $I_1^\sharp$. $I_1^\sharp = T_P^\sharp(I_0^\sharp) = \{\langle a, \mathcal{R}^\sharp_{\vec{v}\mapsto\vec{\lambda}}(\pi^\sharp_\emptyset(\chi_{C1}))\rangle\} = \{\langle a, \phi_1\rangle\}$ where $\phi_1 = \{(\lambda_1 \in \ell(\mathbf{0})),\ (\lambda_2 \in \beta \Leftarrow \lambda_3 \in \beta),\ (\lambda_3 \in \beta \Leftarrow \lambda_2 \in \beta)\}$.

Calculating $I_2^\sharp$.

$$\begin{array}{rcl} I_2^\sharp & = & T_P^\sharp(I_1^\sharp) \\ & = & \left\{ \begin{array}{c} \langle a, \mathcal{R}^\sharp_{\vec{v}\mapsto\vec{\lambda}}(\pi^\sharp_\emptyset(\chi_{C1}))\rangle, \\ \langle a, \mathcal{R}^\sharp_{\vec{v}\mapsto\vec{\lambda}} \circ \pi^\sharp_{\{xs1,zs1,x\}}(\chi_{C2} \odot^\sharp \mathcal{R}^\sharp_{\vec{\lambda}\mapsto\vec{v}'}(\phi_1))\rangle \end{array} \right\} \\ & = & \{\langle a, \phi_1\rangle, \langle a, \phi_2\rangle\} \end{array}$$

where $\phi_2 = \mathcal{R}^\sharp_{\vec{v}\mapsto\vec{\lambda}} \circ \pi^\sharp_{\{xs1,zs1,x\}}(\chi_{C2} \odot^\sharp \mathcal{R}^\sharp_{\vec{\lambda}\mapsto\vec{v}'}(\phi_1))$ is calculated as follows. During the computation, tautologies are removed from an abstract substitution and so is a type dependency that is implied by another in the same abstract substitution. Let $L = \{xs1, zs1, x\}$. We first calculate

$$\begin{array}{cl} \multicolumn{2}{l}{\pi^\sharp_L(\chi_{C2} \odot^\sharp \mathcal{R}^\sharp_{\vec{\lambda}\mapsto\vec{v}'}(\phi_1))} \\ = & \pi^\sharp_L \left(\left\{ \begin{array}{c} (x \in \beta \Leftarrow xs \in \ell(\beta)), \\ (xs1 \in \ell(\beta) \Leftarrow xs \in \ell(\beta)), \\ (xs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge xs1 \in \ell(\beta_2)), \\ (x \in \beta \Leftarrow zs \in \ell(\beta)), \\ (zs1 \in \ell(\beta) \Leftarrow zs \in \ell(\beta)), \\ (zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge zs1 \in \ell(\beta_2)), \\ (xs1 \in \ell(\mathbf{0})), \\ (ys \in \beta \Leftarrow zs1 \in \beta), \\ (zs1 \in \beta \Leftarrow ys \in \beta) \end{array} \right\} \right) \\ = & \pi^\sharp_{\{zs1,x\}} \left(\left\{ \begin{array}{c} (x \in \beta \Leftarrow xs \in \ell(\beta)), \\ (x \in \beta \Leftarrow zs \in \ell(\beta)), \\ (zs1 \in \ell(\beta) \Leftarrow zs \in \ell(\beta)), \\ (xs \in \ell(\beta) \Leftarrow x \in \beta) \\ (zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge zs1 \in \ell(\beta_2)), \\ (ys \in \beta \Leftarrow zs1 \in \beta), \\ (zs1 \in \beta \Leftarrow ys \in \beta) \end{array} \right\} \right) \\ = & \pi^\sharp_{\{x\}} \left(\left\{ \begin{array}{c} (x \in \beta \Leftarrow xs \in \ell(\beta)), \\ (x \in \beta \Leftarrow zs \in \ell(\beta)), \\ (ys \in \ell(\beta) \Leftarrow zs \in \ell(\beta)), \\ (xs \in \ell(\beta) \Leftarrow x \in \beta), \\ (zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge ys \in \ell(\beta_2)) \end{array} \right\} \right) \\ = & \left\{ \begin{array}{c} (xs \in \ell(\mathbf{1})),\ (xs \in \ell(\beta) \Leftarrow zs \in \ell(\beta)), \\ (ys \in \ell(\beta) \Leftarrow zs \in \ell(\beta)), \\ (zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow xs \in \ell(\beta_1) \wedge ys \in \ell(\beta_2)), \\ (zs \in \ell(\mathbf{1}) \Leftarrow ys \in \ell(\beta)) \end{array} \right\} \end{array}$$

Thus, $\phi_2 = \{c_1, c_2, c_3, c_4, c_5\}$ where

$$\begin{array}{rcl} c_1 & = & (\lambda_1 \in \ell(\mathbf{1})) \\ c_2 & = & (\lambda_1 \in \ell(\beta) \Leftarrow \lambda_3 \in \ell(\beta)) \\ c_3 & = & (\lambda_2 \in \ell(\beta) \Leftarrow \lambda_3 \in \ell(\beta)) \\ c_4 & = & (\lambda_3 \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow \lambda_1 \in \ell(\beta_1) \wedge \lambda_2 \in \ell(\beta_2)) \\ c_5 & = & (\lambda_3 \in \ell(\mathbf{1}) \Leftarrow \lambda_2 \in \ell(\beta)) \end{array}$$

Calculating $I_3^\sharp$. It can be shown that $I_3^\sharp = I_2^\sharp$.

The success patterns for a/3 state that when $a(xs, ys, zs)$ succeeds, (1) xs is an empty list and ys and zs are of the same type; or (2) xs is a list (c_1), both xs and ys are lists of elements of type β if zs is (c_2 and c_3), zs is a list of elements of type $\beta_1 \sqcup \beta_2$ if xs is a list of elements of type β_1 and ys is a list of elements of type β_2 (c_4), and zs is a list if ys is (c_5). Let $c_4' = (zs \in \ell(\beta) \Leftarrow xs \in \ell(\beta) \wedge ys \in \ell(\beta))$ and $\phi_2' = \{c_2, c_3, c_4'\}$ where c_1, c_2 are the same as in ϕ_2. The analysis result for the append program in [14] is $\{\langle a, \phi_1 \vee \phi_2'\rangle\}$ which is equivalent to $\{\langle a, \phi_1\rangle, \langle a, \phi_2'\rangle\}$. The result from the new analysis is more precise than that in [14] since ϕ_2 is more precise than ϕ_2' as follows. Firstly, c_1 and c_5 are missing from ϕ_2'. Secondly, if $c' = (\lambda_1 \in \ell(\mathit{int}))$ and $c'' = (\lambda_1 \in \ell(\ell(\mathit{int})))$ then $(zs \in \ell(\mathit{int} \sqcup \ell(\mathit{int})))$ is implied by $\{c', c'', c_4\}$ but not $\{c', c'', c_4'\}$. Hence, c_4 is more precise than c_4'.

5.5 Analysis of Quicksort

The quicksort program has three user-defined predicates. One of them is the append predicate $\texttt{a}/3$ in the previous section. The two others are as follows.

```
pt(V,Xs,Ys,Zs) :- Xs=[ ], Ys=[ ], Zs=[ ].   (C3)
pt(V,Xs,Ys,Zs) :- Xs=[X|Xs1], Ys=[X|Ys1],
                  X<V, pt(V, Xs1,Ys1,Zs).   (C4)
pt(V,Xs,Ys,Zs) :- Xs=[X|Xs1], Zs=[X|Zs1],
                  X>=V, pt(V, Xs1,Ys,Zs1).  (C5)
sort(Xs,Ys) :- Xs=[ ], Ys=[ ].              (C6)
sort(Xs,Ys) :- Xs=[X|Xs1], Ys3=[X|Ys2],
               pt(X,Xs1,Xs2,Xs3),
               sort(Xs2,Ys1),
               sort(Xs3,Ys2),
               a(Ys1,Ys3,Ys).               (C7)
```

Builtin Predicates. The quicksort program uses two builtin predicates $\texttt{<}/2$ and $\texttt{>=}/2$. Builtin predicates are handled by pre-computing success patterns for each builtin predicate and collecting them in the initial iterate $I_0^\sharp$. For instance, $I_0^\sharp = \{\langle \texttt{<}, \{(\lambda_1 \in \mathit{number}), (\lambda_2 \in \mathit{number})\}\rangle, \langle \texttt{>=}, \{(\lambda_1 \in \mathit{number}), (\lambda_2 \in \mathit{number})\}\rangle\}$ for the quicksort program where the builtin type number denotes the set of numbers and will be abbreviated as n. Note that success patterns for builtin predicates are independent of the program to be analyzed and can be pre-calculated and incorporated into an implementation of the analysis.

Analysis result. The analysis is done by first computing the success patterns for $\texttt{a}/3$ and $\texttt{pt}/4$ and then those for $\texttt{sort}/2$. The details of the computation are omitted. The set of success patterns of the quicksort program is

$$I^\sharp = \{\ \langle a, \phi_1\rangle, \langle a, \phi_2\rangle, \langle pt, \phi_3\rangle, \langle pt, \phi_4\rangle, \langle sort, \phi_5\rangle\ \}$$

where ϕ_1 and ϕ_2 are the same as in section 5.4 and

$$\begin{array}{rcl} \phi_3 & = & \{\lambda_2 \in \ell(\mathbf{0}), \lambda_3 \in \ell(\mathbf{0}), \lambda_4 \in \ell(\mathbf{0})\} \\ \phi_4 & = & \left\{ \begin{array}{c} \lambda_1 \in n, \lambda_2 \in \ell(n), \lambda_3 \in \ell(n), \lambda_4 \in \ell(n), \\ \lambda_2 \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow \lambda_3 \in \ell(\beta_1) \wedge \lambda_4 \in \ell(\beta_2), \\ \lambda_3 \in \ell(\beta) \Leftarrow \lambda_2 \in \ell(\beta), \lambda_4 \in \ell(\beta) \Leftarrow \lambda_2 \in \ell(\beta), \end{array} \right\} \\ \phi_5 & = & \left\{ \begin{array}{c} \lambda_1 \in \ell(\mathbf{1}), \lambda_2 \in \ell(\mathbf{1}), \\ \lambda_1 \in \ell(\beta) \Leftarrow \lambda_2 \in \ell(\beta), \lambda_2 \in \ell(\beta) \Leftarrow \lambda_1 \in \ell(\beta) \end{array} \right\} \end{array}$$

The analysis result indicates that when sort(Xs, Ys) succeeds, both Xs and Ys are lists of elements of the same type. The result is precise since queries like sort($[a]$, $[a]$) succeed with the program. Nevertheless, the result is different from the intended use of the quicksort program whereby both Xs and Ys are expected to be lists of numbers. The difference between the inferred and the expected success patterns stems from clause C3 which may succeeds with argument V being any term. If it is modified into

```
pt(V,Xs,Ys,Zs) :- number(V),
                  Xs=[ ], Ys=[ ], Zs=[ ].   (C3')
```

then the analysis result becomes

$$J^\sharp = \{\ \langle a, \phi_1\rangle, \langle a, \phi_2\rangle, \langle pt, \phi_4\rangle, \langle sort, \phi_6\rangle\ \}$$

where

$$\phi_6 = \left\{ \begin{array}{c} \lambda_1 \in \ell(n), \lambda_2 \in \ell(n), \\ \lambda_1 \in \ell(\beta) \Leftarrow \lambda_2 \in \ell(\beta), \lambda_2 \in \ell(\beta) \Leftarrow \lambda_1 \in \ell(\beta) \end{array} \right\}$$

Note that there is now only one success pattern for pt/4 since the success pattern computed from clause C3' is $\langle pt, \{\lambda_1 \in n, \lambda_2 \in \ell(\mathbf{0}), \lambda_3 \in \ell(\mathbf{0}), \lambda_4 \in \ell(\mathbf{0})\}\rangle$ which is subsumed by $\langle pt, \phi_4\rangle$. The inferred success pattern for sort/2 now agrees with the expected one. Type dependencies $\lambda_1 \in \ell(\beta) \Leftarrow \lambda_2 \in \ell(\beta)$ and $\lambda_2 \in \ell(\beta) \Leftarrow \lambda_1 \in \ell(\beta)$ in ϕ_6 capture information that is not expressed by the two other type dependencies in ϕ_6. For example, they inform that if sort/2 succeeds with its first argument being a list of integers then its second argument is also a list of integers, and vice versa.

6. SUMMARY

We have presented a new type dependency inference analysis for logic programs. The analysis infers polymorphic type dependencies from the program and a set of type rules which define types. The use of set union as a type constructor and non-deterministic type definitions enables the analysis to infer more precise type dependencies than other analyses.

The most related work is [14]. The success patterns inferred by [14] are similar to those inferred by the new analysis. An abstract substitution is expressed in [14] as a logic program whose clauses corresponds to type dependencies in the new analysis. The new analysis infers more precise type dependencies than [14] for the following two reasons. Firstly, the new analysis uses set union as a type constructor which is missing from [14]. Lack of set union as a type constructor implies that the join operation on types incurs a loss of precision. Secondly, type definitions are non-deterministic in the new analysis while they are deterministic in [14]. Consequently, the type language in the new type analysis is more powerful and allows more types to be defined than in [14]. The above two characteristics also distinguish the new analysis from those in [2, 6, 7]. For instance, the inferred type for the concrete atom $p([1, [1]])$ is $p(\ell(\mathbf{1}))$ according to [2, 6]. The abstract atom $p(\ell(\mathbf{1}))$ is less precise than the success pattern $\langle p, \lambda_1 \in \ell(int \sqcup \ell(int))\rangle$ inferred by the new analysis. The type constructor $\oplus$ in [7] does not denote set union. For example, the term $[1, [1]]$ has type $\ell(int) \oplus \ell(\ell(int))$ according to [7] while it has type $\ell(int \sqcup \ell(int))$ in the new analysis. Furthermore, type dependencies are not expressed explicitly in clausal form in [2, 6, 7]. Instead, they are captured by type parameters. We contend that an explicit representation of type dependencies in both [14] and the new analysis provides better insight into the behavior of the program for development activities such as debugging.

The type analysis in [16] is a goal-dependent analysis and captures type dependencies between the input and the output states through type parameters. It suffers the same limitation as [2, 6, 7]. The set union as a type constructor and non-deterministic type definitions have been employed in a goal-dependent analysis [17]. The analysis, however, does not trace type dependencies. Type inference analysis can also be performed without given type definitions. An example of such an analysis is [12] that approximates the success set of the program by a unary regular logic program [18]. The analysis infers both type definitions and types and is incorporated into the Ciao System [13]. Type inference analyses that infer type definitions do not trace type dependencies. A notable exception is [5] that captures type dependencies via type parameters. The analysis infers parametric type definitions and type signatures for the predicates in the program so that the program is well typed. The inferred type signatures approximates the set of well typed calls instead of the success set.

7. REFERENCES

[1] G. Amato and F. Scozzari. On the interaction between sharing and linearity. *CoRR*, abs/0710.0528, 2007.

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

[3] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19/20:149–197, 1994.

[4] J. Boye and J. Maluszynski. Two aspects of directional types. In *Proceedings of the Twelfth International Conference on Logic Programming*, pages 747–761. The MIT Press, 1996.

[5] M. Bruynooghe, J. Gallagher, and W. Van Humbeeck. Inference of Well-Typings for Logic Programs with Application to Termination Analysis. In *Proceedings of the Twelveth International Symposium on Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2005.

[6] M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of Prop. In B. Le Charlier, editor, *Proceedings of the First International Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 1994.

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

[8] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007.

[9] 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.

[10] 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.

[11] P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–189. The MIT Press, 1992.

[12] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In M. Bruynooghe, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[13] M. V. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program analysis, debugging, and optimization using the Ciao system preprocessor. In *Proceedings of the 1999 International Conference on Logic Programming*, pages 52–65. The MIT Press, 1999.

[14] P. M. Hill and F. Spoto. Generalising *Def* and *Pos* to Type Analysis. *Journal of Logic and Computation*, 12(3):497–542, 2002.

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

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

[17] L. Lu. Improving precision of type analysis using non-discriminative union. *Theory and Practice of Logic Programming*, 8(1):33–79, 2008.

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

APPENDIX

A. PROOFS

A.0.1 Proposition 4.4

Let $c_1 = (y \in T \Leftarrow \mu_1 \wedge (x \in T_1) \wedge \mu_2)$ and $c_2 = (x \in T_2 \Leftarrow \nu)$ be normalized type dependencies. For any substitution $\eta \in \mathit{Subst}_\sim$, if $\eta \models c_1$ and $\eta \models c_2$ then $\eta \models \mathit{unf}(c_1, c_2)$.

PROOF. Without loss of generality, we assume $\mathsf{Para}(c_1) \cap \mathsf{Para}(c_2) = \emptyset$. Let $\kappa_1 = \mathit{match}(T_1, T_2)$ and $\kappa_2 = \mathit{match}(T_2, T_1)$. There are three cases to consider: case (i) $y \in \mathbf{V}(\nu)$ or $\kappa_i = \diamond$ for $i = 1, 2$; case (ii) $y \notin \mathbf{V}(\nu)$ and $\kappa_1 \neq \diamond$; and case (iii) $y \notin \mathbf{V}(\nu)$ and $\kappa_1 = \diamond$ and $\kappa_2 \neq \diamond$. In the case (i), $\mathit{unf}(c_1, c_2) = (y \in \mathbf{1})$ is a tautology and hence the lemma holds. Now consider the case (ii). Let κ be an arbitrary type valuation such that $\mathit{dom}(\kappa) \supseteq \mathbf{V}(\mathit{unf}(c_1, c_2))$. Then $\mathit{dom}(\kappa) \supseteq \mathbf{V}(c_i) \setminus \{x\}$ for $i = 1, 2$. Assume that $\eta \models \kappa(\mu_1 \wedge \nu \wedge \mu_2)$. Since $\eta \models c_2$ and $\eta \models \kappa(\nu)$, we have that $\eta \models (x \in \kappa(T_2))$ and hence $\eta \models (x \in \kappa \circ \kappa_1(T_1))$. Since c_1 is normalized, $\mathit{dom}(\kappa_1) \cap \mathsf{Para}(\mu_j) = \emptyset$ for $j = 1, 2$. Thus, $\eta \models \kappa \circ \kappa_1(\mu_j)$ for $j = 1, 2$ since $\eta \models \kappa(\mu_j)$ for $j = 1, 2$. Since $\eta \models c_1$, $\eta \models \kappa \circ \kappa_1(\mu_j)$ and $\eta \models (x \in \kappa \circ \kappa_1(T_1))$, we have that $\eta \models \kappa \circ \kappa_1(T)$. Therefore, by the definition of $\models$, $\eta \models (y \in \kappa_1(T) \Leftarrow \mu_1 \wedge \nu \wedge \mu_2)$. Case (iii) is similar to case (ii).

Observe that $\eta \models c$ implies $\eta \models \kappa''(c)$ for any c and κ''. From the definition of *norm*, $\kappa''(c) \leq \mathit{norm}(c)$ for some κ''. So, $\eta \models \mathit{unf}(c_1, c_2) = \mathit{norm}(y \in \kappa'(T) \Leftarrow \mu_1 \wedge \nu \wedge \mu_2)$. □

A.0.2 Theorem 4.6

If $\eta \models \phi$ and $\zeta \models \psi$ then

(a) $(\eta \odot \zeta) \models (\phi \odot^\sharp \psi)$;

(b) $\pi_X(\eta) \models \pi^\sharp_X(\phi)$; and

(c) $\mathcal{R}_{\vec{x} \mapsto \vec{y}}(\eta) \models \mathcal{R}^\sharp_{\vec{x} \mapsto \vec{y}}(\phi)$.

PROOF. • Consider (a) first. We have that $(\eta \odot \zeta) \leq \eta$ and $(\eta \odot \zeta) \leq \zeta$. Let $\eta \models \phi$ and $\zeta \models \psi$. Then $(\eta \odot \zeta) \models \phi$ and $(\eta \odot \zeta) \models \psi$ because types are closed under instantiation. Thus, $(\eta \odot \zeta) \models \phi \cup \psi = \phi \odot^\sharp \psi$.

- Now consider (b). Observe that $\eta \models \psi$ implies $\pi_{\{x\}}(\eta) \models \psi[\![-x]\!]$ for any substitution η, abstract substitution ψ and variable x. Since $\eta \models \phi$, by proposition 4.4, $\eta \models \psi$ where $\psi = \phi \cup \{\mathit{unf}(c_1, c_2) \mid c_1 \in \phi_1^x \text{ and } c_2 \in \phi_2^x\}$ and ϕ_1^x and ϕ_2^x are the same as those in the definition of $\pi^\sharp_x$. Therefore, $\pi_{\{x\}}(\eta) \models \psi[\![-x]\!]$ and hence $\pi_{\{x\}}(\eta) \models \pi^\sharp_x(\phi)$ from which the thesis follows.
- We now prove (c). We first prove that if $\vec{x} \cap \vec{y} = \emptyset$ then $\phi[\vec{y}/\vec{x}] = \pi^\sharp_{\vec{x}}(\alpha_\triangle(\vec{x} = \vec{y}) \odot^\sharp \phi)$. Let $\vec{x} = x_1 x_2 \cdots x_n$ and $\vec{y} = y_1 y_2 \cdots y_n$. Then $\vec{x} = \vec{y}$ stands for $x_1 = y_1, x_2 = y_2, \cdots, x_n = y_n$. This is done be induction.

Basis. n=0. Then $\phi = \phi[\vec{y}/\vec{x}] = \pi^\sharp_\emptyset(\emptyset \odot^\sharp \phi) = \phi$.

Induction. By the definitions of $\pi^\sharp$ and $\alpha_\triangle$ and $\odot^\sharp$, we have

$$\begin{aligned}
&\pi^\sharp_{x\vec{x}}(\alpha_\triangle(x\vec{x} = y\vec{y}) \odot^\sharp \phi) \\
&\quad= \pi^\sharp_x \circ \pi^\sharp_{\vec{x}}(\alpha_\triangle(x = y) \cup \alpha_\triangle(\vec{x} = \vec{y}) \cup \phi) \\
&\quad= \pi^\sharp_x(\alpha_\triangle(x = y) \cup \pi^\sharp_{\vec{x}}(\alpha_\triangle(\vec{x} = \vec{y}) \cup \phi)) \\
&\qquad \text{by the induction hypothesis,} \\
&\quad= \pi^\sharp_x(\alpha_\triangle(x = y) \cup \phi[\vec{y}/\vec{x}]) \\
&\quad= \pi^\sharp_x(\{x \in \beta \Leftarrow y \in \beta, y \in \beta \Leftarrow x \in \beta\} \cup \phi[\vec{y}/\vec{x}]) \\
&\quad= \phi[\vec{y}/\vec{x}][y/x] \\
&\quad= \phi[y\vec{y}/x\vec{x}]
\end{aligned}$$

The case where $\vec{x} \cap \vec{y} \neq \emptyset$ is reduced into the above case by applying (a) and (b).

□

A.0.3 Lemma 5.6

The function γ is a complete meet-morphism.

PROOF. By definition, we have $\gamma(\{\langle p, \phi\rangle \mid p \in \Pi \text{ and } \phi \in \mathsf{TDC}_{\Lambda(p)}\}) = \{\langle p, \eta\rangle \mid p \in \Pi \text{ and } \eta \in \mathit{Subst}_{\Lambda(p)}\}$. It remains to prove that $\gamma(I^\sharp \cap J^\sharp) = \gamma(I^\sharp) \cap \gamma(J^\sharp)$ for any $I^\sharp, J^\sharp \in \mathsf{Int}^\sharp$.

Assume that $\langle p, \eta\rangle \in \gamma(I^\sharp \cap J^\sharp)$. By the definition of γ, there is a ϕ such that $\langle p, \phi\rangle \in (I^\sharp \cap J^\sharp)$ and $\eta \models \phi$. Then $\langle p, \phi\rangle \in I^\sharp$ and $\langle p, \phi\rangle \in J^\sharp$ and hence $\langle p, \eta\rangle \in \gamma(I^\sharp)$ and $\langle p, \eta\rangle \in \gamma(J^\sharp)$. Thus, $\gamma(I^\sharp \cap J^\sharp) \subseteq \gamma(I^\sharp) \cap \gamma(J^\sharp)$.

Assume that $\langle p, \eta\rangle \in \gamma(I^\sharp) \cap \gamma(J^\sharp)$. By the definition of γ, there are ϕ and ψ such that $\langle p, \phi\rangle \in I^\sharp$, $\langle p, \psi\rangle \in J^\sharp$, $\eta \models \phi$ and $\eta \models \psi$. By the definition of $\models$, we have that $\eta \models (\phi \cup \psi)$. Furthermore, $\langle p, (\phi \cup \psi)\rangle \in I^\sharp$ and $\langle p, (\phi \cup \psi)\rangle \in J^\sharp$ since $I^\sharp =\downarrow I^\sharp$ and $J^\sharp =\downarrow J^\sharp$. By the definition of γ, $\langle p, \eta\rangle \in \gamma(I^\sharp \cap J^\sharp)$. So, $\gamma(I^\sharp) \cap \gamma(J^\sharp) \subseteq \gamma(I^\sharp \cap J^\sharp)$, which, together with $\gamma(I^\sharp \cap J^\sharp) \subseteq \gamma(I^\sharp) \cap \gamma(J^\sharp)$, implies $\gamma(I^\sharp) \cap \gamma(J^\sharp) = \gamma(I^\sharp \cap J^\sharp)$. $\square$

A.0.4 Theorem 5.7

The abstract semantics approximates the concrete semantics correctly, i.e., $(\mathit{lfp}\ T_P) \subseteq \gamma(\mathit{lfp}\ T_P^\sharp)$.

PROOF. It suffices to prove that $T_P \circ \gamma(I^\sharp) \subseteq \gamma \circ T_P^\sharp(I^\sharp)$ for an arbitrary $I^\sharp \in \mathsf{Int}^\sharp$ [10]. Assume that $\langle p, \eta\rangle \in T_P \circ \gamma(I^\sharp)$. By the definition of T_P, there are $p \in \Pi$, $(p(\vec{x}_0) \leftarrow E, p_1(\vec{x}_1), \cdots, p_n(\vec{x}_n)) \in P$ and $\langle p_i, \eta_i\rangle \in \gamma(I^\sharp)$ for $i = 1..n$ such that $\eta = \mathcal{R}_{\vec{x}_0 \mapsto \Lambda(p)}(\pi_L(\mathit{mgu}(E) \odot \mathcal{R}_{\Lambda(p_1)\mapsto\vec{x}_1}(\eta_1) \odot \cdots \odot \mathcal{R}_{\Lambda(p_n)\mapsto\vec{x}_n}(\eta_n))$ where $L = (\mathbf{V}(E) \cup \bigcup_{1\leq i\leq n} \vec{x}_i) \setminus \vec{x}_0$. By the definition of γ, there are $\langle p_i, \phi_i\rangle \in I^\sharp$ for $i = 1..n$ such that $\eta_i \models \phi_i$. We also have $\mathit{mgu}(E) \models \alpha_\Delta(E)$. Let $\phi = \mathcal{R}^\sharp_{\vec{x}_0\mapsto\Lambda(p)}(\pi^\sharp_L(\alpha_\Delta(E) \odot^\sharp \mathcal{R}^\sharp_{\Lambda(p_1)\mapsto\vec{x}_1}(\eta_1) \odot^\sharp \cdots \odot^\sharp \mathcal{R}^\sharp_{\Lambda(p_n)\mapsto\vec{x}_n}(\eta_n))$. By theorem 4.6.(a),(b) and (c), we have $\eta \models \phi$. By the definition of $T_P^\sharp$, $\langle p, \phi\rangle \in T_P^\sharp(I^\sharp)$. By the definition of γ, $\langle p, \eta\rangle \in \gamma \circ T_P^\sharp(I^\sharp)$. Therefore, $T_P \circ \gamma(I^\sharp) \subseteq \gamma \circ T_P^\sharp(I^\sharp)$ and hence $(\mathit{lfp}\ T_P) \subseteq \gamma(\mathit{lfp}\ T_P^\sharp)$ according to [10]. $\square$

B. CALCULATION OF $I_3^\sharp$ IN SECTION 5.4

By the definition of $T_P^\sharp$,

$$\begin{array}{rcl}
I_3^\sharp & = & T_P^\sharp(I_2^\sharp) \\
& = & \left\{ \begin{array}{c} \langle a, \mathcal{R}^\sharp_{\vec{v}\mapsto\vec{\lambda}}(\pi^\sharp(\chi_{C1}))\rangle, \\ \langle a, \mathcal{R}^\sharp_{\vec{v}\mapsto\vec{\lambda}} \circ \pi^\sharp_{\{xs1,zs1,x\}}(\chi_{C2} \odot^\sharp \mathcal{R}^\sharp_{\vec{\lambda}\mapsto\vec{v}'}(\phi_1))\rangle, \\ \langle a, \mathcal{R}^\sharp_{\vec{v}\mapsto\vec{\lambda}} \circ \pi^\sharp_{\{xs1,zs1,x\}}(\chi_{C2} \odot^\sharp \mathcal{R}^\sharp_{\vec{\lambda}\mapsto\vec{v}'}(\phi_2))\rangle \end{array} \right\} \\
& = & \left\{ \begin{array}{c} \langle a, \phi_1\rangle, \langle a, \phi_2\rangle, \\ \langle a, \mathcal{R}^\sharp_{\vec{v}\mapsto\vec{\lambda}} \circ \pi^\sharp_{\{xs1,zs1,x\}}(\chi_{C2} \odot^\sharp \mathcal{R}^\sharp_{\vec{\lambda}\mapsto\vec{v}'}(\phi_2))\rangle \end{array} \right\}
\end{array}$$

Let $L = \{xs1, zs1, x\}$. We now calculate

$$\begin{array}{l}
\pi^\sharp_{\{xs1,zs1,x\}}(\chi_{C2} \odot^\sharp \mathcal{R}^\sharp_{\vec{\lambda}\mapsto\vec{v}'}(\phi_2)) \\
\quad = \quad \pi^\sharp_L(\chi_{C2} \odot^\sharp \mathcal{R}^\sharp_{\vec{\lambda}\mapsto\vec{v}'}(\phi_2)) \\
\quad = \quad \pi^\sharp_L \left(\left\{ \begin{array}{c}
(x \in \beta \Leftarrow xs \in \ell(\beta)), \\
(xs1 \in \ell(\beta) \Leftarrow xs \in \ell(\beta)), \\
(xs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge xs1 \in \ell(\beta_2)), \\
(x \in \beta \Leftarrow zs \in \ell(\beta)), \\
(zs1 \in \ell(\beta) \Leftarrow zs \in \ell(\beta)), \\
(zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge zs1 \in \ell(\beta_2)), \\
(xs1 \in \ell(\mathbf{1})), \\
(xs1 \in \ell(\beta_1) \Leftarrow zs1 \in \ell(\beta_1)), \\
(ys \in \ell(\beta) \Leftarrow zs1 \in \ell(\beta)), \\
(zs1 \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow xs1 \in \ell(\beta_1) \wedge ys \in \ell(\beta_2)), \\
(zs1 \in \ell(\mathbf{1}) \Leftarrow ys \in \ell(\beta_2))
\end{array} \right\} \right) \\
\quad = \quad \pi^\sharp_{\{zs1,x\}} \left(\left\{ \begin{array}{c}
(x \in \beta \Leftarrow xs \in \ell(\beta)), \\
(x \in \beta \Leftarrow zs \in \ell(\beta)), \\
(zs1 \in \ell(\beta) \Leftarrow zs \in \ell(\beta)), \\
(xs \in \ell(\mathbf{1}) \Leftarrow x \in \beta), \\
(xs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge zs1 \in \ell(\beta_2)), \\
(zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge zs1 \in \ell(\beta_2)), \\
(zs1 \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow xs \in \ell(\beta_1) \wedge ys \in \ell(\beta_2)), \\
(zs1 \in \ell(\mathbf{1}) \Leftarrow ys \in \ell(\beta_2)), \\
(ys \in \ell(\beta) \Leftarrow zs1 \in \ell(\beta))
\end{array} \right\} \right) \\
\quad = \quad \pi^\sharp_{\{x\}} \left(\left\{ \begin{array}{c}
(x \in \beta \Leftarrow xs \in \ell(\beta)), \\
(x \in \beta \Leftarrow zs \in \ell(\beta)), \\
(xs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow x \in \beta_1 \wedge zs \in \ell(\beta_2)), \\
(xs \in \ell(\mathbf{1}) \Leftarrow x \in \beta_1) \\
(zs \in \ell(\beta_1 \sqcup \beta_2 \sqcup \beta_3) \Leftarrow \left(\begin{array}{c} x \in \beta_1 \\ \wedge \\ xs \in \ell(\beta_2) \\ \wedge \\ ys \in \ell(\beta_3) \end{array} \right)), \\
(zs \in \ell(\mathbf{1}) \Leftarrow x \in \beta_1 \wedge ys \in \ell(\beta_2)), \\
(ys \in \ell(\beta) \Leftarrow zs \in \ell(\beta))
\end{array} \right\} \right) \\
\quad = \quad \left\{ \begin{array}{c}
(xs \in \ell(\mathbf{1})),\ (xs \in \ell(\beta) \Leftarrow zs \in \ell(\beta)), \\
(ys \in \ell(\beta) \Leftarrow zs \in \ell(\beta)), \\
(zs \in \ell(\beta_1 \sqcup \beta_2) \Leftarrow xs \in \ell(\beta_1) \wedge ys \in \ell(\beta_2)), \\
(zs \in \ell(\mathbf{1}) \Leftarrow ys \in \ell(\beta))
\end{array} \right\} \\
\quad = \quad \pi^\sharp_{\{xs1,zs1,x\}}(\chi_{C2} \odot^\sharp \mathcal{R}^\sharp_{\vec{\lambda}\mapsto\vec{v}'}(\phi_1))
\end{array}$$

Therefore, $\mathcal{R}^\sharp_{\vec{v}\mapsto\vec{\lambda}} \circ \pi^\sharp_{\{xs1,zs1,x\}}(\chi_{C2} \odot^\sharp \mathcal{R}^\sharp_{\vec{\lambda}\mapsto\vec{v}'}(\phi_2)) = \phi_2$ and hence $I_3^\sharp = I_2^\sharp$.